Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

EZ-USB HX3PD Hub Firmware User Guide

Document Number: 002-29425 Rev. **

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
www.cypress.com

Contents

1. Getting Started with HX3PD Hub Firmware 5
1.1 Using the Reference Projects 5
1.1.1 Compiling the Project with PSoC Creator 5
2. DMC Firmware Architecture 8
2.1 Firmware Blocks 8
2.2 DMC Firmware Usage Model 9
2.3 Firmware Versioning 10
2.4 Flash Memory Map 11
2.5 Bootloader 12
2.6 Firmware Operation 12
3. Customizing the DMC Firmware Application 14
3.1 Firmware Directory Structure 14
3.2 HX3PD Hub DMC 16
3.2.1 PSoC Creator Schematic 16
3.2.2 Post-Build Script 17
3.2.3 DMC Library 17
3.2.4 Updating Code to Match the Dock Design 18
4. DMC Firmware APIs 22
4.1 API Summary 22
4.1.1 Application Layer API 22
4.1.2 Firmware Update Module API 23
4.1.3 Billboard Module API 24
4.1.4 Host Processor Interface (HPI) API 25
4.1.5 Hardware Adaptation Layer (HAL) API 25
4.1.6 I2C Master Interface API 25
4.1.7 SPI Master Interface API 25
4.1.8 UART Interface API 25
4.1.9 DCSI Interface API 25
4.1.10 Firmware Update API 25
4.1.11 DMC USB API 26
4.1.12 Miscellaneous Configuration APIs 27
4.2 API Usage Examples 27
4.2.1 Boot API Usage 27
4.2.2 GPIO API Usage 28
4.2.3 Timer API Usage 28
4.2.4 Firmware Update Module (I2C/SPI/UART) API Usage 28
5. PD Controller Firmware Architecture 30
5.1 Firmware Blocks 30
5.2 PD Firmware Usage Model 31

5.3 Firmware Versioning 32
5.4 Flash Memory Map 33
5.5 Bootloader 34
5.6 Firmware Operation 35
6. Customizing the Firmware Application 37
6.1 Solution Structure 37
6.2 HX3PD Hub PD PSoC Creator Schematic 39
6.2.1 Updating Code to Match the Schematic 41
6.3 USB-PD Specification Revisions 43
7. PD Firmware APIs 44
7.1 API Summary 44
7.1.1 Device Policy Manager (DPM) API 44
7.1.2 Host Processor Interface (HPI) API 44
7.1.3 Application Layer API 44
7.1.4 Hardware Adaptation Layer (HAL) API 46
7.1.5 Firmware Update API 46
7.1.6 Miscellaneous Configuration API 46
7.2 API Usage Examples 47
7.2.1 Boot API Usage 47
7.2.2 GPIO API Usage 47
7.2.3 Timer API Usage 48
7.2.4 Sleep Mode Control 49
7.2.5 DPM API Usage 49
7.2.6 Solution-Level Examples 52
Revision History 55

1. Getting Started with HX3PD Hub Firmware

1.1 Using the Reference Projects

The EZ-USB® HX3PD Hub reference design provides a set of firmware resources that allows users to build applications using the Power Delivery (PD) Type-C port controllers, and Dock Management Controller (DMC) for the firmware update of HX3PD Hub components. The reference design includes reference projects for target applications that can be used to jump-start application development.

This version of the HX3PD Hub reference design provides the following:

- **CYUSB4347-BZXC_DMC:** This project implements a Dock Management Controller for the dock application using the CYUSB4347-BZXC_DMC device.
- **CYUSB4347-BZXC_PD**: This project implements a dual Type-C port controller for dock applications using the CYUSB4347-ZXC_PD device. This device supports PD 3.0 specification.

Each reference project is provided in the form of a PSoC® Creator™ workspace. The workspace can be opened using PSoC Creator 4.2; the projects can be customized and compiled.

Note: These projects are designed to work with specific devices mentioned above. Changing the target part using Device Selector will cause the firmware build to fail.

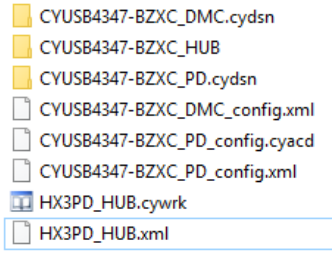
1.1.1 Compiling the Project with PSoC Creator

The **CYUSB4347-BZXC_dmc** project is used to illustrate opening reference projects and building them using PSoC Creator.

1. Navigate to the project folder using Windows Explorer. The project folder contents will look as shown Figure 1-1.

   The **HX3PD_HUB.cywrk** file is the PSoC Creator workspace file that can be opened using the PSoC Creator IDE. If you have installed multiple Creator versions, ensure that the appropriate PSoC Creator version is used to open the workspace.

Figure 1-1. Contents of HX3PD hub Project Folder



Once you open the workspace, there are two projects:

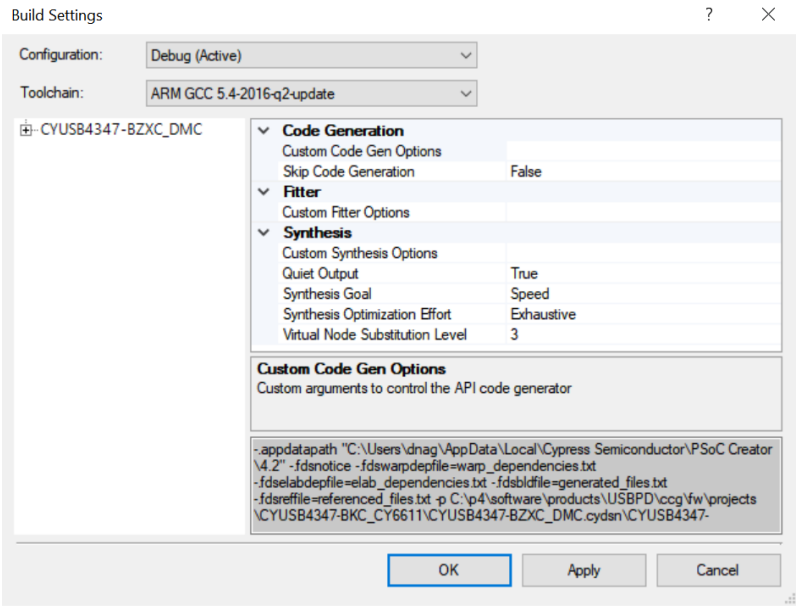
a. **CYUSB4347-BZXC_DMC.cydsn**: The main firmware project for the DMC application. This application is designed to work on top of the bootloader pre-programmed on the DMC device. It also creates two copies of the firmware binary that will be stored in different banks (regions) of the DMC device flash so that the system can implement a fail-safe firmware upgrade mechanism.

b. **CYUSB4347-BZXC_PD.cydsn**: This is the main firmware project for the PD application. This application is designed to work on top of the bootloader pre-programmed on the PD device. It also creates two copies of the firmware binary that will be stored in different regions of the PD device flash so that the system can implement a fail-safe firmware upgrade mechanism.

2. Select **Build** > **Build CY4347USB-BZXC_DMC** or (or, right-click the project name and select the build option).

   Ensure that the compiler toolchain is set to **ARM GCC 5.4-2016-q2-update** (right-click the project and select Build Settings). See Figure 1-2.

Figure 1-2. Project Build Settings



3. Click **Yes** on the pop-up window asking for permission to make the project file writeable.

   The complete build process may take up to three minutes. The output window at the bottom of the IDE will look as shown in Figure 1-4 at the end of the build process.

Figure 1-3. Pop-up Window for Project Write Permissions

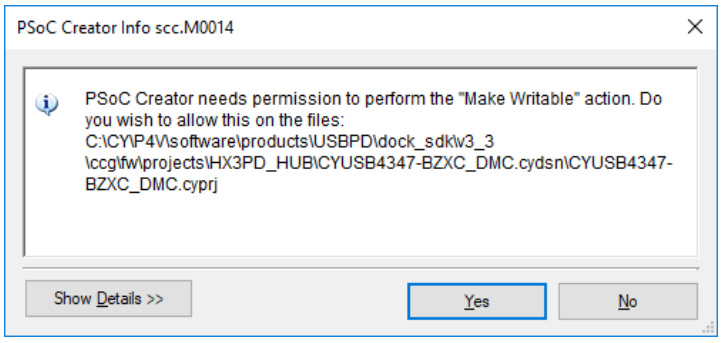


Figure 1-4. Output Window After Build Is Complete

4. Navigate to the project folder to locate the compiled firmware binaries. Navigate to the *CYUSB4347-BZXC_dmc.cydsn\CortexM0\ARM_GCC_541\Debug* folder for the output files. The following three files are the most important output files generated by the build process:

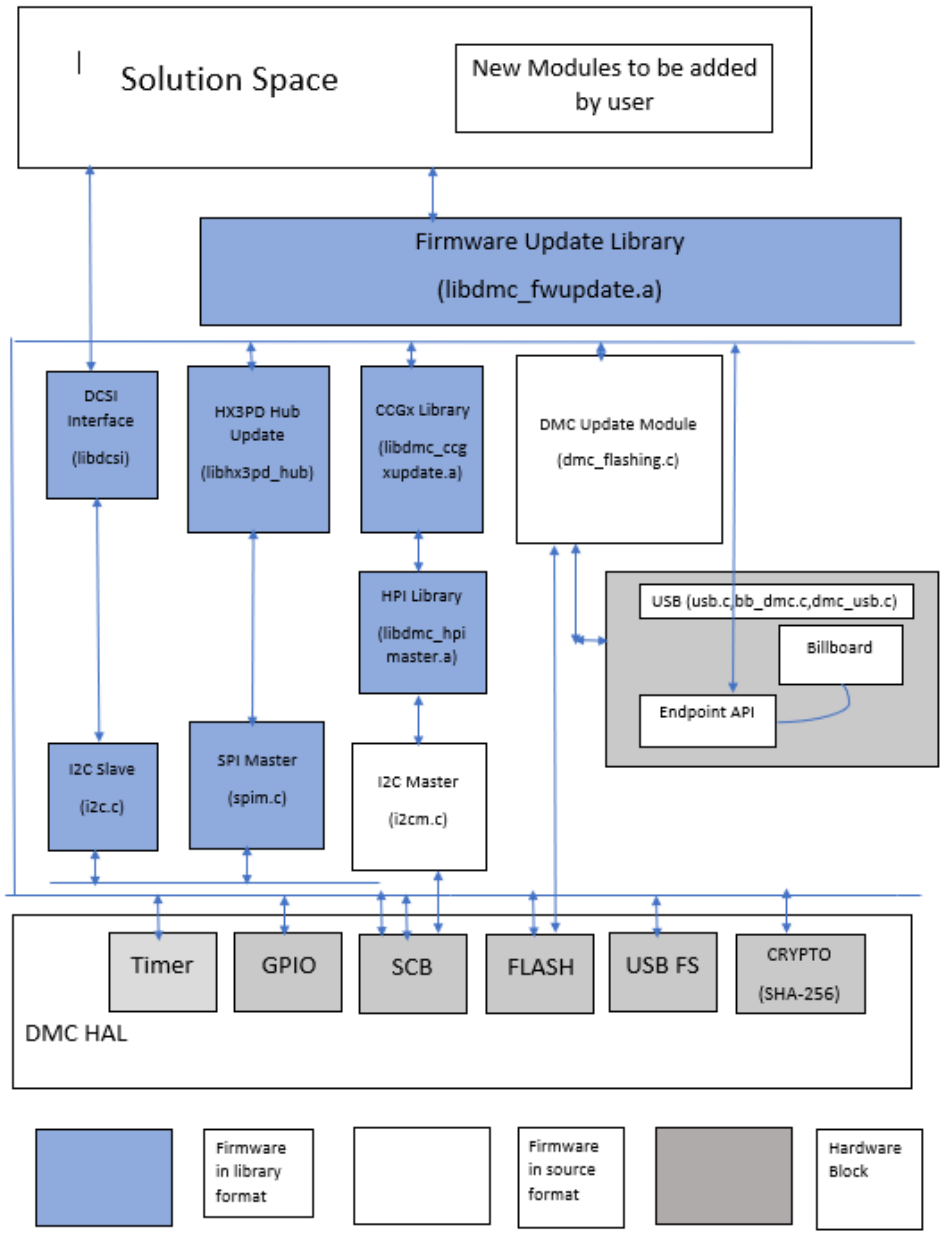
   - ***CYUSB4347-BZXC_dmc_sha.hex***: This is an SWD-programmable binary file in the Intel Hex format that combines the bootloader and both copies of the DMC controller firmware application.
   - ***CYUSB4347-BZXC_dmc_1.cyacd***: This binary file contains the DMC firmware application to be placed in the lower memory bank (region) of the DMC device. The format of the file is documented here.
   - ***CYUSB4347-BZXC_dmc_2.cyacd***: This binary file contains the DMC firmware application to be placed in the upper memory bank of the DMC device.

2. DMC Firmware Architecture

2.1 Firmware Blocks

The DMC firmware is based on a DMC firmware stack and provides programming hooks and interfaces for customers to implement their own update logic for the custom devices.

Figure 2-1. DMC Firmware Block Diagram



The DMC firmware solution contains the following components:

- Hardware Abstraction Layer (HAL): This includes the low-level drivers for hardware blocks on the DMC device. This includes drivers for Crypto, Serial Communication Blocks (SCBs), GPIOs, USB, the flash module, and timer module.
- USB Billboard: This includes the complete USB 2.0 protocol stack and USB Billboard device class.
- I2C Master: Firmware module allows a driver-based I2C master, which contains the common code base for I2C master handling for all SCBs.
- I2C Slave: Firmware module allows a driver-based I2C slave, which contains the common code base for I2C slave handling to all SCBs.
- SPI Master: Firmware module allows a driver-based SPI master, which contains the common code base for SPI master handling to all SCBs.
- UART: Firmware module that allows driver based UART module which contains common code base for UART function handling to all SCBs.
- HPI Master Library: Firmware library that allows the HPI master to communicate with HX3PD and CCGx PD controllers. This library uses the I2C master module.
- Firmware update library: Firmware library that implements the general state machine needed to update the firmware of any device in the dock. In DMC applications, firmware update will be done from the USB host side through the USB vendor interface. It also includes the DMC vendor command handler, which handles the firmware update related vendor commands and dock metadata manager. Dock metadata maintains the information related to the firmware status of dock components.
- DMC Flashing (Update) module: Includes the module-specific code that is required to update the DMC firmware that resides in the DMC Flash. You can refer this module to add firmware update support for any new devices.
- Solution-specific tasks: Includes the application layer code where any custom tasks required by the user can be implemented. This layer provides sufficient hooks where you can also invoke/include the new (custom) modules for updating the firmware of new devices.
- CCGx Library: Includes the module-specific code that is required to update the CCGx firmware that resides in the CCGx Flash over the I2C interface. This module uses the HPI Master Library and I2C Master module to communicate with HX3PD PD controllers.
- HX3PD Hub library: Includes the module-specific code that is required to update the Hub firmware.
- DCSI library : Dock Control and Status Interface (DCSI) for configuration, providing status of ports, and notification of events on the ports from an external embedded controller (EC).
- FGPIO library: Used for configuring, reading, and clearing pins status. For DMC, use the functions in the *dmc_fgpio.h* header file.

Note: Firmware update for devices that you add must be developed based on the DMC Flashing (update) module sample code.

2.2 DMC Firmware Usage Model

1. Load the solution workspace using PSoC Creator.
2. Edit the project schematics and solution configuration header file if required.
3. Use the *ezpd_dockconfiguredmc.exe* utility to build the configuration table, and copy the generated C source file into the PSoC Creator project if necessary.

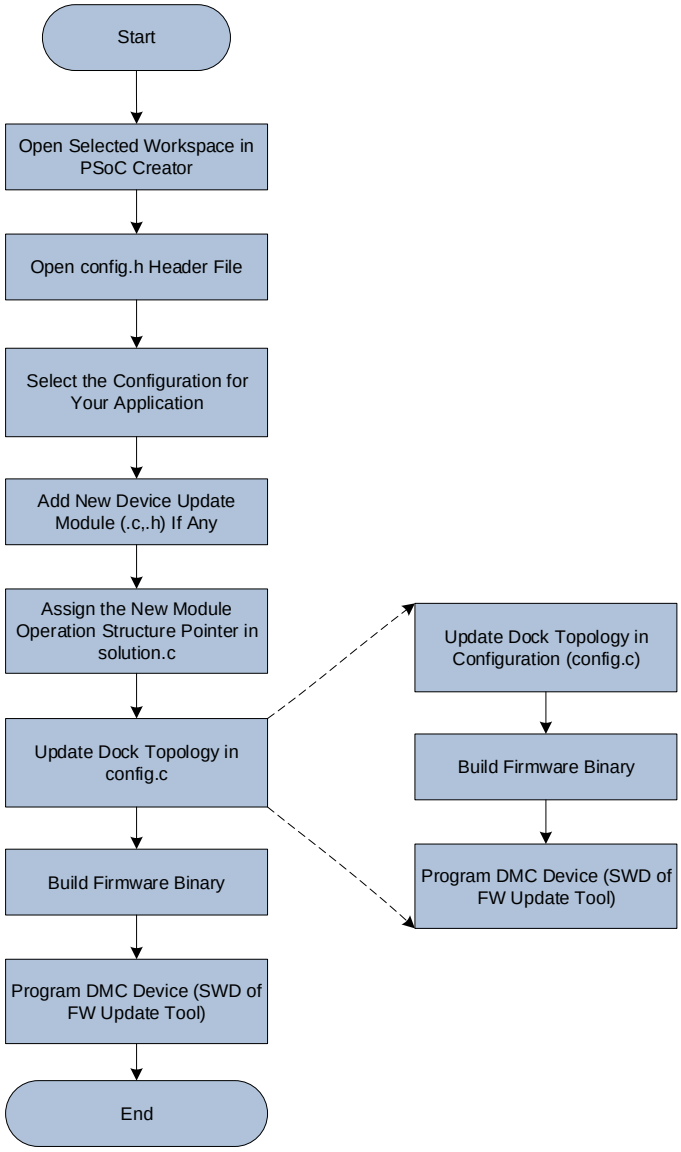
   Alternatively, you can edit the firmware files (.cyacd) directly using the EZ-PD Dock DMC Configuration Generation Tool by specifying the .cyacd files in the XML file. See *HX3PD Hub Reference Design Guide* for details.
4. Build the application projects using PSoC Creator.

   Two types of outputs are created:

   - Firmware binaries in HEX format suitable for SWD programming (using MiniProg)
   - CYACD format that you can use for creating the composite image through the *ezpd_dockcreateimage.exe* utility. You can use the *ezpd_dockfwupdatefw.exe* utility to update the firmware of the DMC.
5. Load the firmware binary onto the target hardware for evaluation and testing. See *HX3PD Hub Reference Design Guide* for details.

Figure 2-2 shows these steps. Many of these steps, such as changing the compile time configurations, are required only if you want to change the way the application works. Use dock configuration tools to change the configuration table whenever the DMC needs to change the dock topology of a new dock design.

Figure 2-2. DMC Firmware Usage Flow



2.3 Firmware Versioning

Each project has a firmware version (base version) and an application version number.

The base firmware version number consists of the major version, minor version, and patch version in addition to an automatically updated build number. It applies to the whole stack and is common for all applications and projects using the stack. See the *src/system/ccgx_version.h* header file.

The application version is modified for individual customers based on requirements. This consists of the major version, minor version, external circuit specification, and application name. You can update this version information as required in the *Firmware/projects/<project_name>/common/app_version.h* header file.

Note: Ensure that you do not change the application name from the value defined for the HX3PD PD application type. The application type information is used by the firmware update mechanism to determine the application/ device type.

The version number information for each firmware is stored in an eight-byte data field.

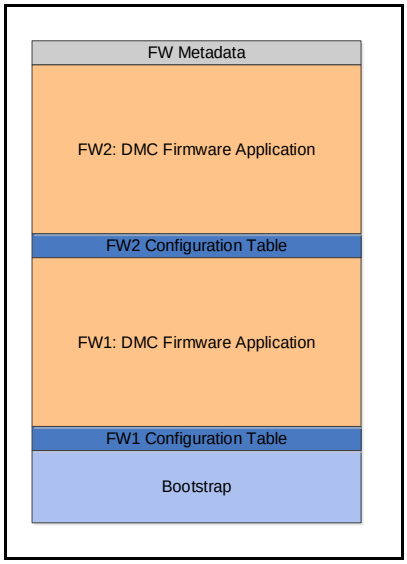
Table 2-1. DMC Firmware Version Structure

Bit Field	Name	Description
[15:0]	Base FW Build Number	Base firmware. Do not edit.
[23:16]	Base FW Patch Version Number	Base firmware patch version number. Cypress updates this field for base firmware releases. Do not edit.
[27:24]	Base FW Minor Version Number	Base firmware minor version number. Cypress updates this field for base firmware releases. Do not edit.
[31:28]	Base FW Major version Number	Base firmware major version number. Cypress updates this field for base firmware releases. Do not edit.
[47:32]	Application Name/Number	Application- or customer-specific changes to be done by the designer. By default, this field has the following values: Notebook: "nb" Power Adapter: "pa" Alternate Mode Adapter (AMA): "aa" Dock Management Controller: "dm" **Note**: This information is used by Cypress tools to determine the application type. and should not be modified for standard applications.
[55:48]	External Circuit Number	User-editable application- or customer-specific value.
[59:56]	Application Minor Version Number	DMC firmware minor version number for DMC stack firmware releases. Do not edit.
[63:60]	Application Major Version Number	DMC firmware major version number for DMC stack firmware releases. Do not edit.

2.4 Flash Memory Map

DMC has a 128-KB flash memory to store a bootloader, and two copies of the firmware binary along with the corresponding configuration table. The flash memory map for the device is shown in Figure 2-3.

Figure 2-3. DMC Flash Memory Map



The bootstrap code in the DMC is used to only load one of the (latest) DMC application firmware images and cannot be used to upgrade the DMC application firmware. It is allocated a fixed area and uses 3 KB of memory. This memory area can only be written to from the SWD interface.

The configuration table holds the default DMC configuration with the dock topology table for the dock application and Billboard-related parameters. It is located at the beginning of each firmware binary; the size of each configuration table is 2 KB.

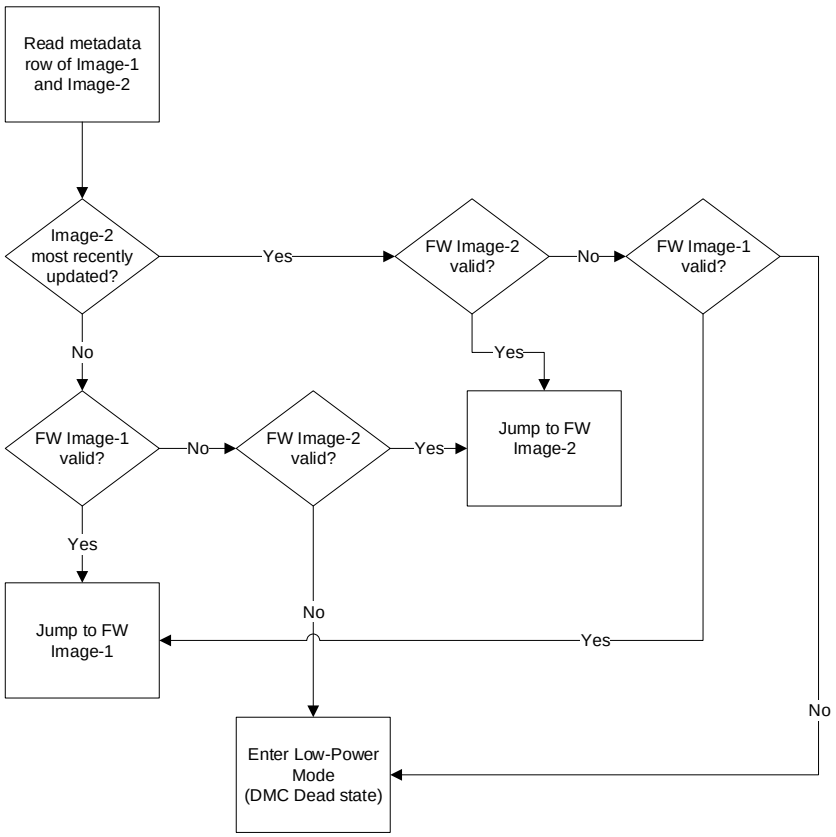
The DMC firmware area is used for the main DMC firmware application. In all applications, two copies of the firmware (called FW1 and FW2) are used. FW1 uses the space from 3 KB to 64 KB; FW2 uses the remaining space.

The metadata area holds the metadata about the firmware binaries. The firmware metadata follows the definition provided by the PSoC Creator bootloader Component, and includes firmware (SHA-256) checksum, size, and start address.

2.5 Bootloader

The flash-based bootloader mainly functions as a bootstrap and is the starting point for firmware execution. It validates the firmware based on SHA-256 checksum stored in flash. The bootstrap doesn't include the flashing module which is used to update the firmware to other devices in the DMC.

Figure 2-4. Bootloader Flow Diagram

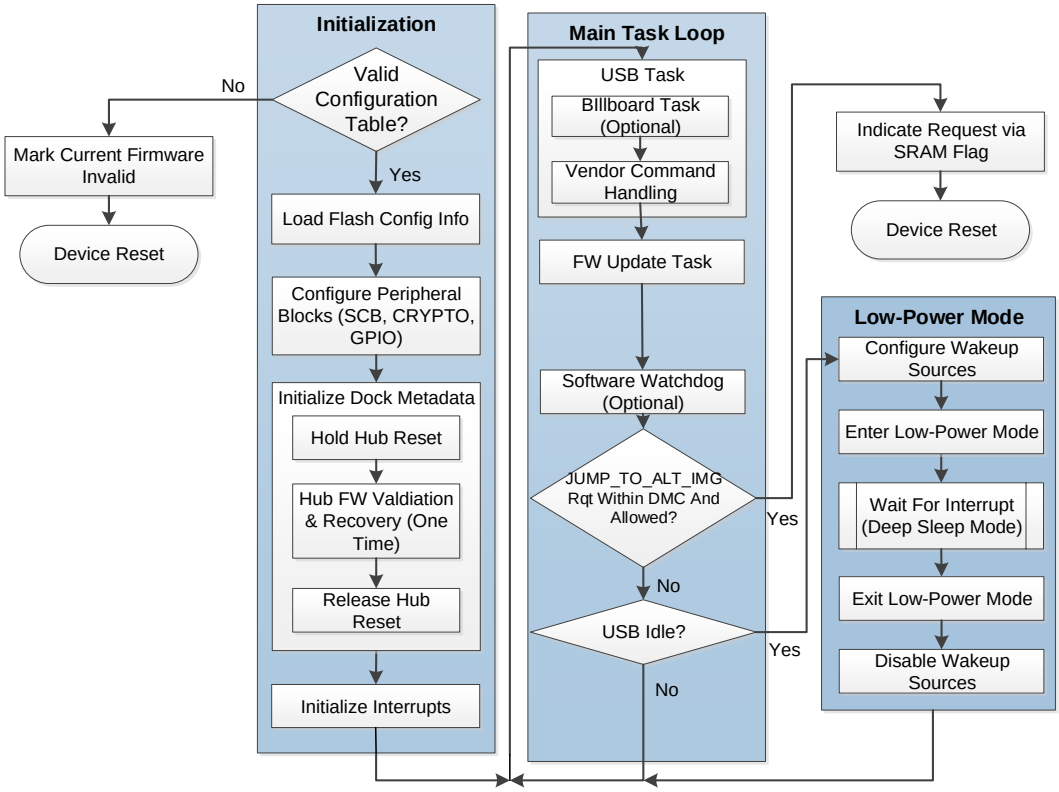


Because the DMC uses redundant firmware images that can update each other, it is expected that the device always has at least one functional image that can be booted by the bootloader. The bootloader keeps track of the last updated firmware image through the metadata, and loads it during startup.

2.6 Firmware Operation

Figure 2-5 shows the firmware initialization and operation sequence. The DMC firmware is implemented in the form of a set of state machines and tasks that need to be performed periodically.

Figure 2-5. DMC Firmware Flow Diagram



The code flow for the application is implemented in the *common\main.c* file. As can be seen from the main () function, the implementation is a simple round-robin loop, which services each of the tasks that the application has to perform. See the EZ-USB *HX3PD Hub's DMC Firmware API Guide* for more details of these functions and handlers.

3. Customizing the DMC Firmware Application

As explained in *EZ-USB HX3PD Hub Design Guide*, a DMC configuration can be modified without changing the firmware source. The DMC configuration includes the Composite Dock Topology Table (CDTT) information and Billboard configuration parameters. See *EZ-USB HX3PD Hub Reference Design Guide* for more details on the configuration parameters and to understand how to change the DMC configuration.

Any addition of new device types to the composite dock topology or any other custom changes will, however, require changes to the firmware sources implementing the application. This chapter walks you through the process of updating the firmware implementation.

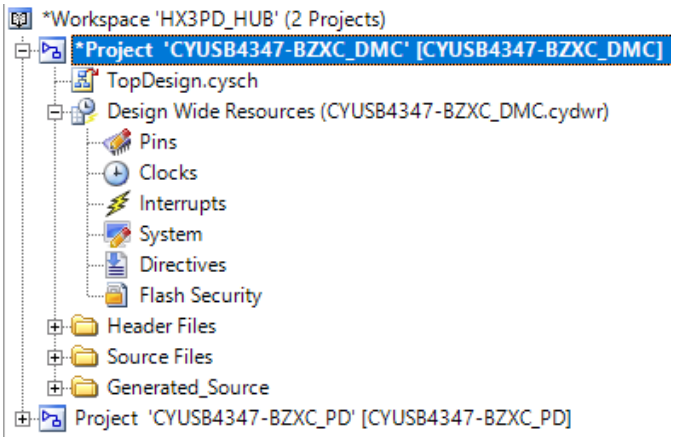
Note: As the firmware sources and reference projects are installed in the Firmware folder, do not make changes to the original installed version of these files. You can create a copy of the *Firmware* folder from the SDK installation, and use the copy for making any changes. This will ensure that you have a clean version of the files that you can revert to as well.

Because the target application remains the same, it is expected that the changes are limited to aspects such as the mechanism for adding a new device type to the composite dock topology, designing for signed and non-signed FW update. This does not involve changes to the core functionality implemented by the DMC device.

3.1 Firmware Directory Structure

The DMC solution structure is shown in Figure 3-1 with the CYUSB4347-BZXC_dmc workspace as reference. Source and header files used in the solution are grouped into different folders.

Figure 3-1. DMC Solution Structure



- *crypto*: The *crypto* folder contains the header and source files providing Crypto Hardware IP Abstraction layer. Do not modify header file definitions because these are used by the DMC stack library and conflicting definitions can result in undefined behavior.
- *dmc*: The *dmc* folders contain header and source files that implement the firmware update logic for different devices. In addition, this folder contains the files necessary for USB enumeration and for the device to act as a Host Processor Interface (HPI) master. If any new device is added to the topology or if the FW update logic is changed for any existing device, you should add the related files here. The source files include the following:

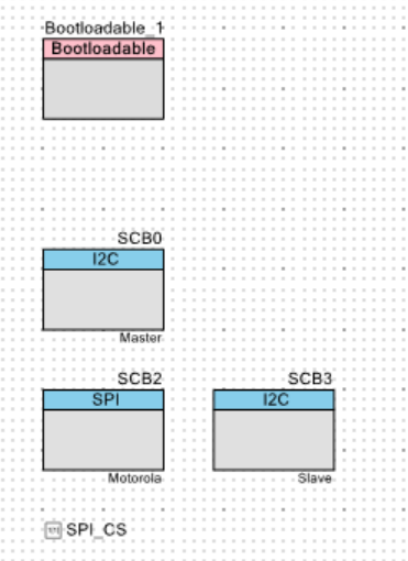
- *billboard.h* : Contains Billboard control interface information
- *ccgx_update.h*: Provides the FW update logic and hardware interface information of the CCGx device
- dmc.h: Contains common definitions and structures used in the DMC
- *dmc_app.h, dmc_app.c*: Contain the declaration for application-level functions
- *dmc_flashing.h, dmc_flashing.c*: Provide the FW update logic and hardware interface information of the DMC device
- *dmc_sha.h, dmc_sha.c*: Contain the function declaration and definition for the SHA-256 algorithm
- *dmc_solution.h, dmc_solution.c*: Solution-layer header and source files for the DMC application
- *dmc_usb.h, dmc_usb.c*: Contain the USB interface related typedefs and function declarations
- *fw_update.h*: Contains the structures and helper functions used for the FW update logic implemented by the DMC
- *hx3pd_hub_intf.h,hx3pd_hub_update.h***:** Contain the structures and function definitions required for hub updates
- *hpi_master.h*: CCGx HPI master interface header file
- *i2cm.h, i2cm.c*: I2C master interface header file and source files
- *uart.h, uart.c*: UART interface header file and source files
- *spim.h, spim.c*: SPI master interface header file and source files
- *spi_eeprom_master.h, spi_eeprom_master.c*: SPI master interface header file and source files to the SPI EEPROM slave
- *bb_dmc.c*: Billboard interface source file for the DMC
- *dmc_fgpio.h*: This header file is used for configuring, *reading*, and clearing pins status

- ***Solution***: The *solution* folder contains header and source files that provide user configurations, stack parameters, and application version information. The solution-level sources include the following:
  - *config.h*: Enables/disables firmware features and provides the composite version information
  - *stack_params.h*: Contains silicon ID information along with the timer module configuration
  - *app_version.h, ccgx_version.h*: Defines the DMC app and base firmware version
  - *instrumentation.h*: Contains the definitions associated with functions to implement high-level instrumentation supported by the DMC application to track task execution latencies and runtime stack usage.
  - *instrumentation.c*: Contains application-level instrumentation code
  - *flash_config.h*: Defines THE flash configuration for the DMC device
  - *config.c*: Contains the default runtime configuration for the DMC application that has been generated using the HX3PD Utility tools
  - *cyapicallbacks.h*: Provides macro callback for code generated by PSoC Creator. For more information, see the “Macro Callbacks” topic in the PSoC Creator Help.
  - *main.c*: Contains the main application entry point
- ***system***: This folder contains the base system-level functionality such as GPIO, soft timer implementation, flash driver, and firmware upgrade handlers. Do not modify the header file definitions because these are used by the DMC and HPI stack libraries; conflicting definitions can result in undefined behavior.
- ***usb***: This folder contains the base system-level functionality for the USB protocol. Do not modify the header file definitions because these are used by the DMC and HPI stack libraries; conflicting definitions can result in undefined behavior.

3.2 HX3PD Hub DMC

3.2.1 PSoC Creator Schematic

Figure 3-2. PSoC Creator Schematic for DMC



Double-click the schematic (*TopDesign.cysch)* file to open the schematic editor window (see Figure 3-2). The schematic shows how the internal resources of the DMC device are used in the design. This includes various serial communication blocks and the bootloadable block. You can add available GPIO pins to communicate with external elements if required.

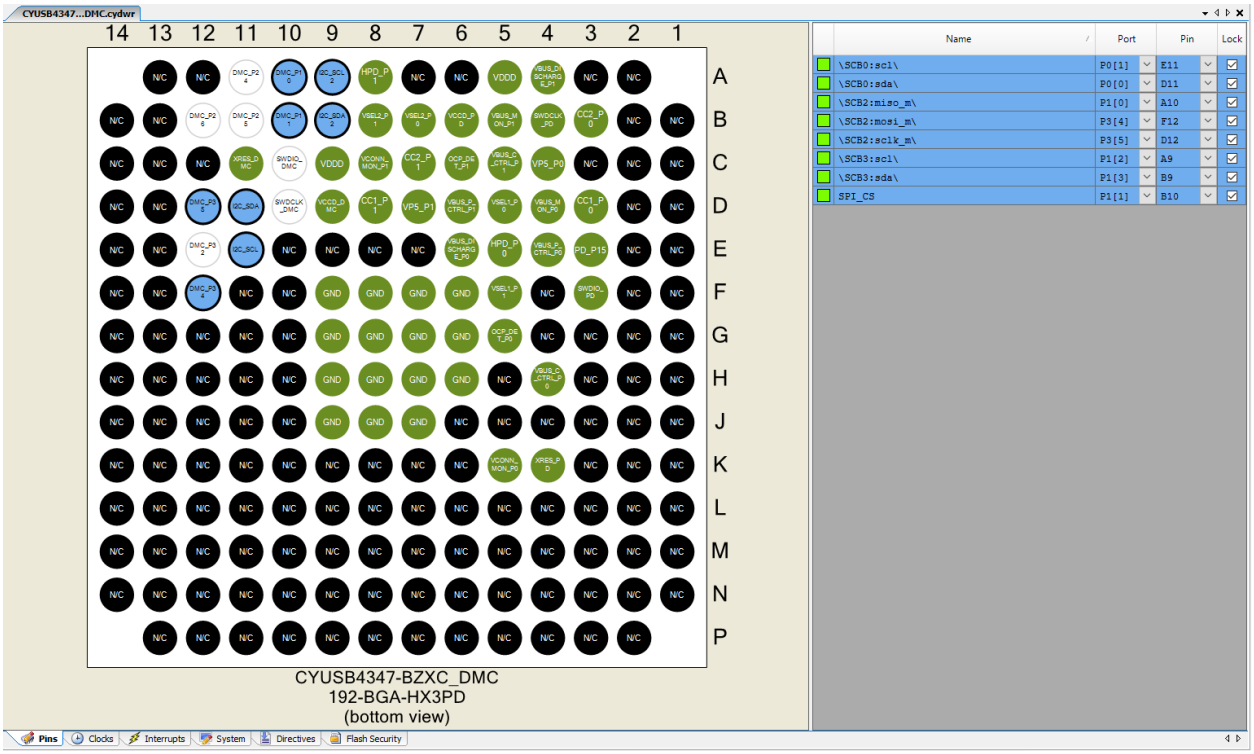
Table 3-1 shows the schematic elements used in the DMC project. The selection of some of these elements is fixed due to the capabilities of the DMC device and the bootloader design. The table also points out the changes allowed in the schematic design.

Table 3-1. Schematic Elements in HX3PD Hub's DMC Design

Schematic Element	Description	Changes Allowed
Bootloadable_1	Software block that interacts with the bootloader on the DMC device	No
SCB0	Serial communication block (SCB) used to connect peripherals	No
SCB2	SCB used to connect peripherals	No
SCB3	SCB used to connect peripherals	No
SPI_CS	Chip Select line used for SPI transactions	No

Closely associated with the schematic is the Design Wide Resources (DWR) view, which maps each schematic element to a pin, clock, or hardware block on the DMC device. Open the *CYUSB4347-BZXC_DMC.cydwr* file to see the DWR settings for the project.

Figure 3-3. DWR Project Settings



As shown in Figure 3-3, the DWR view has several tabs, which configure pin mapping, interrupt mapping, clock selection, flash security, and so on. It is recommended that you restrict any changes to the DWR to the pin mapping view. Do not change the clock, interrupt, system, or flash configurations. Even in the pin mapping editor, changes should be subject to the constraints outlined in Figure 3-3.

3.2.2 Post-Build Script

As part of build process, PSoC Creator will invoke the *post_build* script present in the project folder. The post-build script in-turn invokes the *hex_bin_update.exe* tool in the .\ *hex_bin_update* folder. It requires the dll file, pyd file and base library files present in the same folder. Do not change the post-build script or any files in the .\ *hex_bin_update* folder.

3.2.3 DMC Library

All the DMC related libraries are present in the *lib* folder along with *projects* and *src*. This folder contains the following libraries:

- *libdmc_hpimaster.a*: Mandatory library for the DMC to upgrade the firmware of the supported CCGx devices over I2C. This library must not be removed from project linker settings. This library enables the device to be the HPI master to communicate with supported CCGx devices.
- *libdmc_ccgxupdate.a*: Mandatory library for the DMC to enumerate itself as Billboard and implement the firmware update logic to supported CCGx devices. Do not remove this library from project linker settings. This library enables the Billboard interface and provides the interface functions to update CCGx devices.
- *libdmc_fwupdate.a*: Mandatory library that implements the firmware upgrade state machine. This supports only unsigned update.
- *libhx3pd_hub.a*: Library module used for communication and upgrade of the Hub through the DMC in the HX3PD hub design.
- *libdcsi.a*: Library module that implements the configuration, and provides the port status and event notification on ports from an external embedded controller (EC).

3.2.4 Updating Code to Match the Dock Design

3.2.4.1 Compile-Time Options

The DMC controller application supports a set of features that can be enabled/disabled using compile time options. These options are set in the *config.h* header file that you can find under the *solution* folder.

Table 3-2. Compile-Time Options for DMC Application

Option	Description	Values
SYS_DEEPSLEEP_ENABLE	Flag to enable the DMC deep sleep feature to save power. If this feature is enabled, the DMC will go to deep sleep if all the following conditions are satisfied: ○ No US interrupts are pending ○ DMC is not currently updating the FW ○ No HPI event is pending from devices connected over the HPI bus ○ USB is in Suspend state If this feature is enabled, the DMC will go to sleep if all the following conditions are satisfied: ○ DMC is not currently updating the FW ○ No HPI event is pending from devices connected over the HPI bus ○ USB is in Idle state	1 for Deep Sleep enable 0 for Deep Sleep disable
RESET_ON_ERROR_ENABLE	Enables the instrumentation task to reset the DMC on error conditions (watchdog expiry or hard fault)	1 for Error Recovery enable 0 for Error Recovery disable
APP_FW_LED_ENABLE	Enable flag for the firmware activity LED indication. When enabled, the user LED blinks at 1-second intervals; the user switch on the RDK design hardware cannot be used at this time. This LED can be used for development support but is recommended to be left in the OFF state to save power in production designs. **Note**: Enabling this flag enables a timer to run at 1-second interval, which causes the LED to toggle at every occurrence of timer expiry.	1 for LED enable 0 for LED disable

3.2.4.2 Updating the Default Configuration

The DMC firmware project has an embedded default configuration that specifies the Composite Dock Topology Table (CDTT) information matching the EZ-USB HX3PD Hub Reference Design hardware and default Billboard configuration parameters in the *common\config.c* file. See the *EZ-USB HX3PD Hub Reference Design Guide.*

The contents of the configuration file (*config.c*) can be replaced with that of the .c source file generated by HX3PD Utility tools. After all the source changes are completed, rebuild the project to generate customized binaries.

3.2.4.3 Adding New Device to the Device Topology

Addition of any new device or removal of any device from the existing dock design would need the following steps:

1. Updating the dock topology in DMC configuration (and other configuration parameters, if any) using the EZ-PD Dock DMC Configuration Generation Tool. See EZ-USB HX3PD Hub Design Guide. Edit the *config.c* file in PSoC Creator Source Editor with the contents from the generated C source file by EZ-PD Dock DMC Configuration Generation Tool.
2. After the configuration (dock topology) is updated to include the new device, add support for new device update in the DMC firmware as follows:
   - If the new device is to be updated using I2C as the interface protocol between the DMC and the device, use the existing I2C master driver code (*i2cm.c, i2cm.h*). If SPI or UART protocols are used, use the appropriate driver module (*spim.c/spim.h* or *uart.c/uart.h*) and API functions.

- Device module support: For any new device types along with DMC, supported PD Controller and HX3PD Hub, add the firmware module support to implement the new device's update. The new module should handle the following:
  - Fill the `dmc_update_opern` structure with the function pointers implementing the FW update logic for the new device. See Section 3.2.4.4.
  - Add the source file implementing the added functions under the *dmc* folder in the PSoC Creator solution structure.
  - Initialize `gl_opern` with the newly filled structure for the respective component ID in the `dmc_init_hw_interface()` function. Update the `init_dock_reset()` function for the newly added device in the CDTT.
  - If the new device is connected over HPI I2C, and if any of the HPI events need to be handled, add the event handler in the `sln_hpi_event_handler()` function defined in *main.c*.

    Notes:

    - HPI is the Host Processor Interface protocol over I2C defined by CCGx devices for communication between any EC and CCGx.
    - Any HPI event occurring while firmware update is in progress will be handled by DMC after the firmware update is completed.

3. After support for the device module code is added and integrated with the DMC solution module, rebuild the project to generate DMC firmware binaries.

   Note: Firmware update for the devices that you add must be developed based on the DMC Flashing (update) module sample code. The following example demonstrates using the DMC Flashing (update) module.

Figure 3-4 shows the operation structure with function pointers for the DMC Flashing Module. The *dmc_flashing.c* file implements each of these functions per the DMC flashing requirements.

Figure 3-4. DMC Flashing Module Operation Structure

```
static const dmc_update_opern dmc_operation = {
    dmc_init_dev_param,
    NULL,
    NULL,
    dmc_prepare_update,
    dmc_flash_row_write,
    NULL,
    dmc_check_fw_version,
    dmc_jump_to_alternate,
    dmc_is_dev_query_deferred,
    dmc_update_logic,
    dmc_skip_jump_to_alt_request,
    NULL
};
```

Figure 3-5 shows the code snippet that initializes `gl_opern` in the `dmc_init_hw_interface` function for the DMC device type.

Figure 3-5. DMC Flashing Module Operation Structure Assignment

```
switch (topology->device_type)
{
    case DMC_DEV_TYPE_DMC_CY7C65219:
        gl_opern[comp_id] = get_dmc_operation();
        break;
```

3.2.4.4 Module Operation Structure

Figure 3-6 shows the skeleton of the module operation structure (defined in *dmc_solution.h*). Table 3-3 describes the role of each function pointer in the structure.

Figure 3-6. Module Operation Structure

```
/**
 * @brief Structure holding function pointers of various device modules.
 * This will be filled with comp_id as index - same order as devices are in CDTT.
 * @warning Below listed function pointers shalln not be left NULL.
 * when new device module is being added else the code will brick or may
 * function in an unpredicted manner.
 * init_dev_param
 * flash_row
 * check_fw_version
 * get_cur_image
 */
typedef struct
{
    /*! Mandatory: Device parameters init handler. */
    void (*init_dev_param) (const dev_topology_t *topology);
    /*! Firmware update context init handler. */
    void (*init)(const dev_topology_t *topology);
    /*! Firmware update context deinit handler. */
    void (*deinit)(void);
    /*! Firmware update prepare handler. */
    ccg_status_t (*prepare_update)(image_type_t img_type, bool *deferred, dmc_completion_callback cb);
    /*! Mandatory: Flash row write handler. */
    ccg_status_t (*flash_row)(image_type_t img_type, uint16_t row_id, uint8_t *buffer, uint16_t size, bool *deferred, dmc_completion_c

    /*! FW update process finish handler. */
    ccg_status_t (*finish_update)(image_type_t img_type, bool flashing_status, bool *deferred, dmc_completion_callback cb);
    /*! Mandatory FW version check function. */
    bool (*check_fw_version)(uint8_t *img_version, image_type_t img_type, uint8_t comp_id, bool check_last_valid);
    /*! Jump to alternate image handler. */
    void (*jump_to_alternate)(const dev_topology_t *topology, dmc_completion_callback cb);
    /*! API to defer device status query during initialization sequence */
    bool (*is_dev_query_deferred)(void);
    /*! API to determine the update logic specific for device module based on the image mode of the device and current running image.
    bool (*dev_update_logic)(uint8_t comp_id, image_type_t img_type, bool *jump_to_alt_request);
    /*! API to determine the jump to alternate image requesr specific for device module based on the image mode of the device and cur
    bool (*skip_jump_to_alt_request)(uint8_t comp_id, image_type_t cur_img);

    /*! Configures HX3 related SCB and GPIOs like reset & WP GPIO */
    void (*dev_config_hw_interface)(const dev_topology_t *topology);
}dmc_update_opern;
```

Table 3-3. Module Operation Structure Details

Function Pointer	Description	Reference Code
`init_dev_param`	Queries the respective device and updates the dock metadata RAM copy with the information related to the current running image, image validity, and firmware version of the images. `set_image_status()` and `update_versions()`update the information in the RAM copy. This function pointer cannot be left NULL in the device module operation structure.	`dmc_init_dev_param()` in *dmc_flashing.c*
`init`	Initializes the firmware update context variables, if any, used during firmware update. This function must be invoked by the DMC state machine before starting the firmware update to the particular device. This function pointer can be left NULL in the device module operation structure if no initialization is required.	
`deinit`	De-initializes the firmware update context variables, if any, being used during firmware update. This function must be invoked by the DMC state machine after the firmware update to the particular device is done and before starting the firmware update to another device. This function pointer can be left NULL in the device module operation structure if no de-initialization is required.	
`prepare_update`	Prepares the device for the flashing operation. This function sends any preparatory commands to be sent to the device prior to starting the image update to the device. This must be invoked by the DMC state machine prior to starting the actual image update. This function can be implemented in a blocking or non-blocking manner using the parameter `*deferred`. Set `*deferred = false` if the function is blocking. Set `*deferred = true` if it is non-blocking. In this case, the function callback passed as a parameter to this function must be invoked upon completing the function execution to indicate the completion of the function execution to the DMC state machine. This function pointer can be left NULL in the device module operation structure if no preparatory commands are required.	`dmc_prepare_update()` in *dmc_flashing.c*. This is an example for blocking implementation with `*deferred = false`.

Function Pointer	Description	Reference Code
`flash_row`	Writes the row data received from DMC state machine to the device. This function must be invoked by the DMC state machine for every row, with the appropriate row number and row data along with it. This function can be implemented in a blocking or non-blocking manner using the parameter `*deferred`. Set `*deferred = false` if the function is blocking. Set `*deferred = true` if it is non-blocking. In this case, the function callback passed as a parameter to this function should be invoked upon completing the function execution to indicate the completion of the function execution to the DMC state machine. This function pointer cannot be left NULL in the device module operation structure.	`dmc_flash_row_write()` in *dmc_flashing.c*. This is an example for blocking implementation with `*deferred = false`.
`finish_update`	Implements any finishing tasks that need to be performed for the device following an image update. This function will be invoked by the DMC after the last row of data had been successfully written to the device or in the case of any image write failure to the device. The `flashing_status` parameter passed to the function will indicate if the image update to the device had been completed successfully or otherwise. This function will be invoked after every image update to the device. This function can be implemented in a blocking or non-blocking manner using the parameter `*deferred`. Set `*deferred = false` if the function is blocking. Set `*deferred = true` if it is non-blocking. In this case, the function callback passed as a parameter to this function should be invoked upon completing the function execution to indicate the completion of the function execution to the DMC state machine. This function pointer can be left NULL in the device module operation structure if no finish update tasks are required.	
`check_fw_version`	Implements the logic for the firmware version check for restricting the update of any image updates conditionally based on the firmware version of the incoming new image and existing image in the device. This function pointer can be left NULL in the device module operation structure if no firmware version restriction need to be imposed on device update.	`dmc_check_fw_version()` in *dmc_flashing.c*
`jump_to_alternate`	Implements the logic for initiating the jump to alternate image for the device, if required. This function pointer can be left NULL in the device module operation structure if not required.	`dmc_jump_to_alternate()` in *dmc_flashing.c*
`is_dev_query_deferred`	Defers the calling of `init_dev_param` if the device module need more time for initialization. If deferred, `init_dev_param` will be called when the HX3PD_Status_Query_Script tool is invoked to query the dock status prior to a firmware update. This function pointer can be left NULL in the device module operation structure if the device module needs no delay and `init_dev_param` call need not be deferred.	`dmc_is_dev_query_deferred()` in *dmc_flashing.c*
`dev_update_logic`	Implements the update logic specific for the device module based on the image mode of the device and current running image. This function pointer cannot be left NULL in the device module operation structure. If left as NULL, the device update will not be attempted by the DMC state machine.	`dmc_update_logic` in *dmc_flashing.c*
`skip_jump_to_alt_request`	Implements the logic specific for device module, based on the image mode of the device and current running image, before invoking the jump to alternate request. This function pointer cannot be left NULL in the device module operation structure. If left as NULL, jump to alternate request to the device update will not be attempted by the DMC state machine.	`dmc_skip_jump_to_alt_request()` in *dmc_flashing.c*
`dev_config_hw_interface`	Configures the hardware specific for the device module (like the SCB and GPIOs needed for the device firmware update)	

4. DMC Firmware APIs

This section provides a summary of the API functions provided by the DMC stack and other layers in the in EZ-USB HX3PD Hub's DMC firmware solution. Only the functions that are expected to be used directly from user code are documented here. See the *EZ-USB HX3PD Hub DMC API Guide* for more details on the data structures used and all functions.

4.1 API Summary

4.1.1 Application Layer API

These functions are used by the DMC in the application layer to perform application- or solution-specific tasks. See *src/dmc/dmc_app.h* and *src/dmc/dmc_solution.h* header files.

Table 4-1. Application Layer API Functions

Function	Description	Parameter
`app_init()`	Performs application-level initialization required for the DMC solution	None
`app_task()`	Handler for application-level asynchronous tasks	None
`app_sleep()`	Checks whether the application handlers are ready to allow device deep sleep	None
`app_wakeup()`	Restores the APP handler state after the DMC device wakes from deep sleep	None
`system_sleep()`	Places the DMC device in power saving deep sleep mode if possible. The function checks for each interface being idle and then enters sleep mode with the appropriate wakeup triggers. If the device enters sleep mode, the function will return only after the device has woken up.	None
`is_us_connected()`	Returns the current status of the US connection	None
`dmc_init()`	Initializes the devices available in the dock	None
`dmc_check_fw_version()`	Determines whether an update is required based on the FW version of the image entry in FWCT and the existing image entry in the dock metadata	`img_version`: Image version retrieved from the FWCT `img_type`: Indicates whether the check needs to be done for the `image_type_t` structure `comp_id`: Component Id in the dock `check_last_valid`: Flag to check whether the last updated image is valid.
`dmc_version_check_enabled()`	Determines whether to perform a version check should be done	None
`init_dock_reset()`	Initiates all devices reset: US CCGx reset; then initiates DMC soft reset	None
`us_ccgx_soft_reset_cb()`	Callback registered for initiating a DMC soft reset to be initiated after the successful completion of the US CCGx device's reset	Status: Ignored in this function.

4.1.2 Firmware Update Module API

These API functions are used by the DMC to update the firmware for various HX3PD Hub modules like the DMC,PD Controller and the hub . See the *src/dmc/fw_update.h* header file. The following API functions are top-level API functions; see the *EZ-USB HX3PD HUB DMC API Guide* for a detailed list of API functions.

Table 4-2. Firmware Update API Functions

Function	Description	Parameter	Returns
`init_dock_md()`	Initializes the dock metadata. When the dock metadata is empty in the DMC flash, fetches the dock metadata by querying various devices. When the dock metadata is valid, copies the dock metadata contents from the flash to the RAM copy. Updates the dock metadata in RAM from CDTT parameters and writes back the RAM copy into the flash.	None	None
`get_dev_topology()`	Gets the device topology information from the CDTT based on `comp_id`	`comp_id`: Unique number for which the device topology requires the component ID of the device to be updated	Returns: Device topology when component id is less than the device count; NULL otherwise
`dmc_soft_reset()`	Initiates a soft reset of the DMC device	None	None
`reset_dmc_state()`	Resets the DMC to a known state	None	None
`is_dmc_idle()`	Returns if the DMC FW update state machine has not started and the DMC is in IDLE state	None	Returns true if the dmc is in idle state; false otherwise
`check_img_validity()`	Checks the validity of the image identified by the image type and component id of the device	`comp_id`: Unique number for which the device topology requires `img_type`: This is `image_type_t enum` `img_update_count_ptr`: Pointer to count the update attempts for the image	Returns the status validity of the image which is an intger
`is_image_update_needed()`	Checks if the specific image of the device specified by the component ID needs to be updated, based on FW version check	comp_id: Unique number for which the device topology requires img_type: This is image_type_t enum	Returns true if the umage needs to be updated; false otherwise
`dmc_fw_update_task()`	Handles the firmware update for connected devices and the DMC. This is called as part of the main loop.	None	None

4.1.2.1 DMC Firmware Update API

These API functions are used to download the firmware for the DMC module. See the *src/dmc/dmc_flashing.h* header file. The following API functions are top-level API functions.

Table 4-3. DMC Firmware Update API Functions

Function	Description	Parameter
`get_dmc_operation()`	Gets the pointer to the structure of function pointers related to various DMC operations to perform the firmware download for the DMC module	None

See Section 3.2.4.5 for DMC firmware update module structure and operations.

4.1.2.2 CCGx Firmware Update API

These API functions are used to download the firmware for the CCGx module. See the *src/dmc/ccgx_update.h* header file. The following API functions are top-level API functions.

Table 4-4. CCGx Firmware API Functions

Function	Description	Parameter
`get_ccgx_operation()`	Gets the pointer to the structure of function pointers related to various DMC operations to perform the firmware download for the CCGx part.	None
`ccgx_intf_init()`	Initializes the CCGx HPI interface for DMC operation	None
`ccgx_soft_reset()`	Initiates a soft reset of CCGx	`cb`: Completion callback function.
`ccgx_config_hw_interface()`	Configures CCGx-related SCB and HPI Interrupt GPIO	`topology`: Pointer to the device topology node of the corresponding CCGx device
`ccgx_intf_handle_hpi_event()`	Handles pending HPI events	None
`get_ccgx_reset_ptr()`	Gets the pointer to the `ccgx_soft_reset` structure	None
`retrieve_usb_self_powered_status()`	Retrieves USB self-powered status from the CCGx module	None

See Section 3.2.4.5 for DMC firmware update module structure and operations.

4.1.3 Billboard Module API

The DMC controller has a USB Billboard interface and vendor interface. These API functions are used to control the DMC controllers's Billboard interfaces. See the *src/dmc/billboard.h* header file and the *EZ-USB HX3PD HUB DMC API Guide*.

Table 4-5. Billboard API Functions

Function	Description	Parameter
`bb_update_all_status ()`	Updates the alternate mode status for all modes. The current Billboard implementation supports a maximum of 8 alternate modes. Each mode, as defined in the order of Binary Object Device Store (BOS) descriptor, has two bit status. • Bit 1:0 indicates status of `alt_mode0` • Bit 3:2 indicates status of `alt_mode1` Use this function only when the Billboard status needs to be re-initialized to a specific value. In individual entry / exit cases, use `bb_update_alt_status()`.	`status`: Status data for all alternate modes `addl_failure`: Additional failure info
`usb_bb_get_inf_dscr()`	Retrieves the interface descriptor for the USB Billboard interface	`buffer`: Pointer to the interface descriptor to be copied `max_length`: Maximum length that can be copied to the buffer `inf_num`: Interface number that is copied at the `INTF_DSCR_INF_NUM_OFFSET` offset in the buffer `inf_str_index`: Interface string index that is copied at the `INTF_DSCR_INF_STR_OFFSET` offset in the buffer
`bb_get_bos_dscr ()`	Retrieves the Billboard BOS descriptor	`dscr_len`: Length of BOS descriptor is copied to this buffer. Should be a valid buffer.
`handle_bb_hpi_event()`	Handles the HPI Billboard event.	`response_data`: Buffer size must be minimum of three bytes.
`bb_enum_with_altmode_status()`	Eenumerates Billboard with alternate mode status	None

4.1.4 Host Processor Interface (HPI) API

These API functions are provided by the HPI firmware module. See the *src/hpiss/hpi.h* file for details. These API functions are explained in the *EZ-USB HX3PD HUB DMC API Guide*. Contact Cypress for access to detailed Host Processor Interface (HPI) documentation.

4.1.5 Hardware Adaptation Layer (HAL) API

These API functions provided as part of the Hardware Adaptation Layer (HAL), which provides drivers for hardware blocks on the PD controller. See the *EZ-USB HX3PD Hub DMC API Guide* for details.

4.1.5.1 GPIO API

The PSoC Creator GPIO Component and associated API functions can be used. However, the firmware also provides a set of special API functions for reduced memory footprint. These functions are defined in the *src/system/gpio.c* file.

4.1.5.2 Flash API

The flash API provides the core functionality used for DMC controller configuration and firmware updates. These are wrappers over the PSoC Creator-provided flash APIs. In addition, these wrappers implement checks to ensure that the firmware binary is not corrupted by writing while it is being accessed. The flash-related APIs are defined in *src/system/flash.c*.

4.1.5.3 Timer API

The DMC controller firmware stack uses a software timer implementation for timing measurements with 1 ms granularity; it uses a single hardware timer. If the timer block used is Watchdog Timer (WDT), the timers can be used across device sleep modes. It is also possible to use a tickless implementation, which reduces interrupt frequency. The soft-timer-related API functions are defined in *src/system/timer.c*.

4.1.6 I2C Master Interface API

The Serial Communication Block (SCB) Component in PSoC Creator can be used. However, the firmware provides a dedicated I2C master mode driver, which will be used to communicate and update the firmware for I2C slave components. These API declarations are provided in *src/dmc/i2cm.h*.

4.1.7 SPI Master Interface API

The DMC Firmware provides a dedicated SPI master mode driver, which will be used to communicate and update the firmware for SPI slave components. These API declarations are provided in *src/dmc/spim.h* and *src/dmc/spi_eeprom_master.h*.

4.1.8 UART Interface API

The DMC Firmware provides a dedicated UART driver interface, which will be used to communicate and update firmware for UART components. These API declarations are provided in *src/dmc/uart.h*.

4.1.9 DCSI Interface API

This module consists of the function relating to the (Dock Controller and Status Interface) DCSI interface that can be used on the HX3PD Hub. See the *src/dmc/dcsi_intf.h* for the relevant APIs.

4.1.10 Firmware Update API

DMC controller application support firmware updates through interfaces I2C, UART and API. The firmware update APIs are common functions that are used by each of these protocol modules to implement the firmware update functionality. The firmware update related API are defined in *src/system/boot.c*.

4.1.11 DMC USB API

The DMC controller has a USB Billboard interface and vendor interface. These API functions are used to control the DMC controllers' s USB interfaces. These API declarations are provided in *src/dmc/dmc_usb.h*.

Table 4-6. DMC USB API Functions

Function	Description	Parameter
`dmc_usb_init()`	Initializes the DMC USB module	None
`dmc_usb_enable()`	Queues the DMC USB enumeration/re-enumeration.	None
`dmc_usb_disable()`	Disables the DMC USB device and disconnects the terminations. For internal implementation of Billboard, the USB module is controlled from `dmc_usb_task()`. This function only queues the request. It should be noted that only one pending request is honored. If more than one request is queued, only the latest is handled. A disable call clears any pending enable.	`force`: Determines whether to force a disable. `false` = Interface not disabled when in flashing mode. `true` = Interface disabled regardless of the operation mode
`dmc_cfg_bb_enabled()`	Checks the configuration information and identifies if a Billboard is enabled	None
`dmc_usb_is_present()`	Checks the USB state information and identifies if a DMC USB device exists	None
`dmc_usb_is_idle()`	Returns the status of the DMC USB module (idle or not). This function indicates whether there are any pending tasks. This function can be invoked before the device enters sleep mode to check if it is allowed. However, idle condition does not allow deep sleep entry. For this, the application should use the `dmc_usb_enter_deep_sleep()` function.	None
`dmc_usb_enter_deep_sleep()`	Puts the module into deep sleep. The function first checks whether deep sleep mode can be supported at this time and enables the module to wake up from deep sleep. In this case, the wakeup source is USB. Deep sleep is allowed only when the USB bus is in suspend state. So it is better to check for the USB state first before this call. Failure on this call does not require any special action to be taken by the caller other than not to enter the deep sleep mode until a subsequent call passes successfully.	None
`dmc_usb_task()`	Handles the DMC USB module task. The function implements the DMC USB state machine and needs to be invoked in the main task loop. The task handler allows deferring interrupts and improves interrupt latency of the system	None
`dmc_usb_send_status()`	Load status data into the interrupt endpoint	`ep_index`: Index of the endpoint which will be used for sending DMC status. `data`: Buffer which holds data that needs to be sent over Interrupt endpoint.
`dmc_usb_receive_bulk_data()`	Read the bulk data, if received, into a buffer	`ep_index`: Index of the endpoint to be used for reading bulk data. `buffer_p`: Buffer to which data from endpoint buffer is copied. `ep_len`: The user has to make sure this is valid pointer and it stores the number of bytes received.
`usb_vendor_get_inf_dscr()`	The function retrieves interface descriptor for the USB vendor interface	`buffer`: Vendor interface descriptor is copied to this buffer. `max_length`: Minimum length that is filled into the buffer. `inf_num`: Interface number. `inf_str_index`: Value that is copied into buffer at an offset `INTF_DSCR_INF_STR_OFFSET`
`usb_vendor_inf_ctrl()`	The function enables / disables the vendor bridge mode.	`enable`: Flag to enable or disable the vendor interface.

4.1.12 Miscellaneous Configuration APIs

HX3PD has a set of APIs to configure pins to read the status of certain interrupts and the bus of the hub. The following table provides the description of the configuration API functions defined in *src/dmc/dmc_fgpio.h*.

Table 4-7. Configuration API Functions

Function	Description	Parameter	Return Value
`dmc_spi_status_pin_config()`	Configures the pin to check the status of the SPI bus used by the hub	None	None
`dmc_spi_status_read_value()`	Reads the status of the SPI bus used by the hub	None	True: Pin is HIGH. False: Any other state
`dmc_i2c_int_hub_pin_config()`	Configures the I2C interrupt status pin used by the hub	None	None
`dmc_i2c_int_hub_read_value()`	Reads the status of the I2C interrupt pin used by the hub	None	True: The pin is HIGH. False : Any other state
`dmc_i2c_int_hub_clear_intr()`	Clears the I2C interrupt pin used by the hub	None	None
`dmc_i2c_int_hub_enable_intr()`	Configures the I2C interrupt pin to be used as a desired interrupt mode	`Int_mode`: Desired Interrupt mode `Isr`: Callback address	None
`dmc_i2c_int_hub_get_intr()`	Reads the status of the hub's I2C interrupt pin	None	True: Interrupt is set. False: Not set
`dmc_vbus_in_pin_config()`	Configures the VBUS pin of the hub as digital input	None	None
`dmc_vbus_in_read_value()`	Reads the status of the hub's VBUS pin	None	True: Pin is HIGH. False: Other states
`dmc_vbus_in_clear_intr()`	Clears the pin interrupt flag from the hub's VBUS pin	None	None
`dmc_vbus_in_enable_intr()`	Configures the hub's VBUS pin with the desired interrupt setting	`Int_mode`: Desired Interrupt mode `Isr`: Callback address	None
`dmc_vbus_in_get_intr()`	Reads the status of the hub's VBUS interrupt pin	None	True: Interrupt is set. False: Not set

4.2 API Usage Examples

This section provides a few examples for the usage of the APIs documented. See the *EZ-USB HX3PD Hub DMC API Guide* for more details.

4.2.1 Boot API Usage

The communication of the bootloader and firmware application in the Dock SDK is built using the PSoC Creator Bootloader and Bootloadable Components. This section shows how the PSoC Creator Bootloader and Bootloadable Components along with the wrapper APIs in the SDK work to transfer control from the application firmware to the bootloader or to the application in the alternate memory bank.

4.2.1.1 Perform Device Reset

Because the order in which the bootloader prioritizes firmware images is fixed, resetting the device causes the device to boot back into the same mode that it previously was in. The `CySoftwareReset()` API function can be used to initiate a DMC controller device reset.

```
/* Include relevant header files. */
#include <project.h>

void init_dmc_reset (void)
{
    /* Initiate dmc reset. */
```

```
    CySoftwareReset ();
}
```

4.2.2 GPIO API Usage

APIs provided by the PSoC Creator Pins Component can be used in DMC firmware solutions. In addition to these, specific APIs to perform common GPIO functions are provided in the DMC firmware. See Section 7.2.2.

4.2.3 Timer API Usage

The DMC firmware provides a soft timer module, which can be used for task scheduling. The timer APIs allow users to create one-shot timer objects with callback notification on timer expiry.

Soft timers are identified by using a single-byte timer ID. The caller should ensure that the timer ID used does not collide with timers used elsewhere. This is facilitated by reserving the timer ID range from 0x1E to 0x3C for use by the user application code. These timer IDs are not used internally within the DMC firmware stack and are safe for use.

A soft timer is started using the `timer_start()` API function and can be aborted using the `timer_stop()` API function. See Section 7.2.3 for an example.

4.2.4 Firmware Update Module (I2C/SPI/UART) API Usage

The DMC firmware provides I2C, SPI, and UART module interfaces for the Component to update the firmware and communications on the dock. In the HX3PD Hub, you can use the I2C, SPI, and UART interface to update the Component which has SPI and UART interfaces.

4.2.4.1 I2C Interface API Usage

I2C read and write can be performed using the `i2cm_reg_read()` and `i2cm_reg_write()` API functions. Before using these, make sure that `i2cm_start()`is called to enable the SCB as the I2C Master for Bridge operation. The following example shows how to read and write to EEPROM with the I2C interface.

```
#define EEPROM_SLAVE_ADDRESS   (0x51u)
#define EEPROM_ADDRESS_SIZE    (0x02u)
#define SCB_INDEX          (0x02u)
#define EEPROM_PAGE_SIZE       (0x64u)

uint8_t read_from_eeprom()
{
    uint8_t status;
    uint8_t eeprom_buffer[EEPROM_PAGE_SIZE];
    uint16_t eep_page_addr = 0x0000;

    /* Read from EEPROM */
    status  = i2cm_reg_read (SCB_INDEX, /* SCB Index */
           EEPROM_SLAVE_ADDRESS,   /* EEPROM Slave address */
           eeprom_buffer,      /* Buffer to read from EEPROM */
           EEPROM_PAGE_SIZE,       /* Bytes to read */
           (uint8_t *)(&eeprom_page_addr),/* EEPROM address to read */
           EEPROM_ADDRESS_SIZE);   /* EEPROM address size */

    return status;
}

uint8 write_to_eeprom()
{
uint8_t status;
    uint8_t eeprom_buffer[EEPROM_PAGE_SIZE] = {0x0};
    uint16_t eep_page_addr = 0x0000;

    /* write to EEPROM */
    status  = i2cm_reg_write (SCB_INDEX,/* SCB Index */
           EEPROM_SLAVE_ADDRESS,   /* EEPROM Slave address */
           eeprom_buffer,      /* Buffer to read from EEPROM */
           EEPROM_PAGE_SIZE,          /* Bytes to read */
           (uint8_t *)(&eeprom_page_addr),/* EEPROM address to read */
```

```
          EEPROM_ADDRESS_SIZE);      /* EEPROM address size */

    return status;
}
```

4.2.4.2 UART Interface API Usage

UART read and write can be performed using the `uart_write_data()` and `uart_read_data ()` API functions. Before using these, make sure that `uart_start()` is called to enable the UART in the SCB. The following example shos how to use UART API functions to read and write data.

```
#define SCB_INDEX          (0x02u)

uint8_t write_to_uart(uint8_t * data, uint16_t data_size)
{
               uint8_t status;

    /* Clear RX buffer before transmitting the data. The receiver may send the
     * response at anytime. Its better we clear the RX buffer
     */
       uart_scb_clear_rx_buffer (SCB_INDEX);

    /* Write the data to the UART */
    status = uart_write_data (
              SCB_INDEX,    /* SCB Index */
              data,     /* UART data */
              data_size);   /* Data size */

    return status;
}

void read_from_uart(uint8_t * read_buffer, uint16_t bytes_to_read)
{
    uint8_t status;
    uint8_t data_available = 0;

    /* Wait until the required number of bytes are received in the FIFO */
    do
    {
       /* read the number of bytes available in the FIFO */
       data_available = uart_scb_get_rxd_bytes (SCB_INDEX);
    }while(data_available < bytes_to_read);

    /* Read the data from UART */
    status = uart_read_data (
              SCB_INDEX,       /* SCB Index */
              read_buffer,     /* UART buffer to read the data */
              bytes_to_read);  /* Number of bytes to read */

    return status;
}
```

4.2.4.3 SPI Interface API Usage

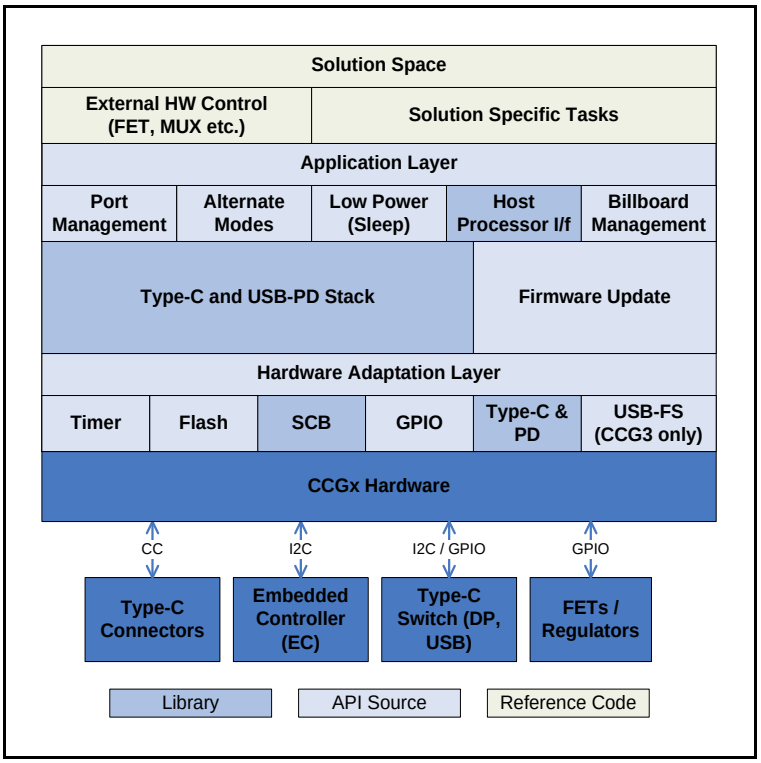
See the spi_eeprom_master.c file in the DMC project firmware to learn about the SPI API to read and write the data to EEPROM.

5. PD Controller Firmware Architecture

5.1 Firmware Blocks

The firmware architecture allows you to implement a variety of USB-PD applications using PD controller devices and a fully tested firmware stack.

Figure 5-1. PD Controller Firmware Block Diagram



The PD controller firmware architecture contains the following components:

- **Hardware Adaptation Layer (HAL):** This includes the low-level drivers for hardware blocks on the CCGx device. This includes drivers for the Type-C and USB-PD block, Serial Communication Blocks (SCBs), GPIOs, flash module, timer module, and USB Full Speed device module (only for CCG3 Dongle).
- **USB Type-C and USB-PD Protocol Stack:** This is the complete USB-PD protocol stack that includes the Type-C and USB-PD port managers, USB-PD protocol layer, the USB-PD policy engine, and the device policy manager. The device policy manager is designed to allow all policy decisions to be made at the application level, either on an external Embedded Controller (EC) or in the CCG firmware itself.
- **Firmware update module:** This is a firmware module that allows the device firmware maintained in internal flash to be updated. In Notebook PD port controller applications, the firmware update will be done from the EC side through an I2C interface.

- **Billboard Management:** This module handles all the billboard enumeration sequences and is applicable only for CCG3 Dongle implementations. The Billboard module is used internally by the Alternate Modes modules and is not expected to be invoked explicitly.
- **Host Processor Interface (HPI):** HPI is an I2C-based control interface that allows an Embedded Controller (EC) to monitor and control the USB-PD port on the CCG device. HPI is the means to allow the PC platform to control PD policy management.
- **Port Management:** This module handles all PD port management functions including the algorithm for optimal contract negotiations, source and sink power control, source voltage selection, port role assignment, and swap request handling.
- **Alternate Modes:** This module implements the alternate mode handling for CCG as a DFP and UFP. A fully tested implementation of DisplayPort alternate mode with CCGx. The module also allows users to implement their own alternate mode support in both DFP and UFP modes. (Not used in HX3PD PD controller)
- **Low Power:** This module attempts to keep the CCG device in the low-power standby mode as often as possible to minimize power consumption.
- **External Hardware Control:** This is a hardware design-dependent module, which controls the external hardware blocks such as FETs, regulators, and Type-C switches.
- **Solution-specific tasks:** This is an application layer module where any custom tasks required by the user solution can be implemented.

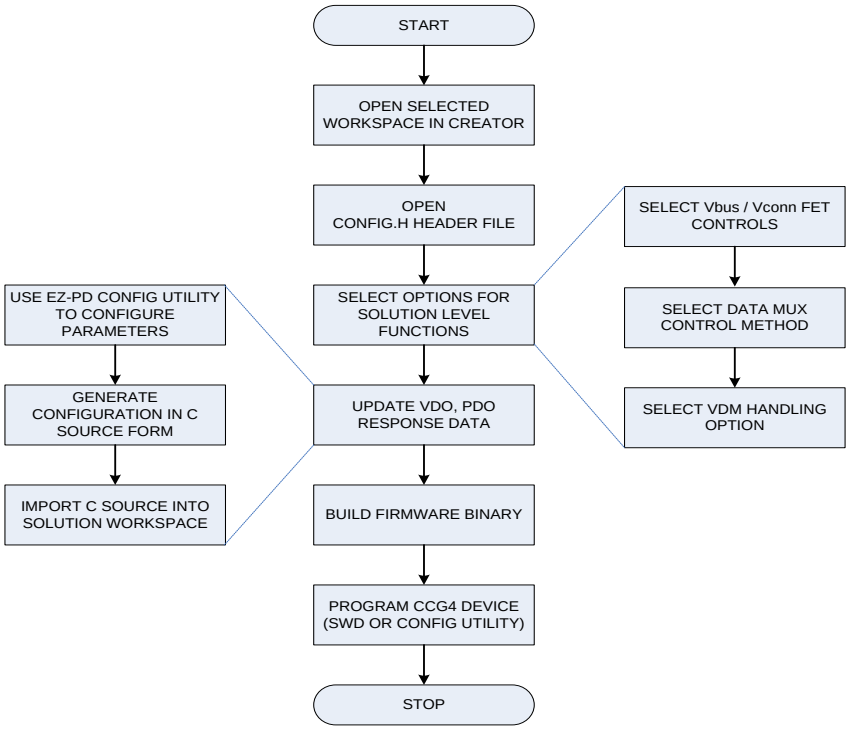
5.2 PD Firmware Usage Model

See the usage flow Figure 5-2:

1. Load the solution workspace using PSoC Creator.
2. Edit the project schematics and solution configuration header file if needed.
3. Use the EZ-PD Configuration Utility to build the configuration table, and copy the generated C source file into the PSoC Creator project if necessary. The configuration table can also be updated by editing the *config.c* file in PSoC Creator Source Editor.
4. Build the application projects using PSoC Creator. The firmware binaries will be generated in ELF, HEX, and CYACD formats suitable for SWD programming, PSoC MiniProg, and the EZ-PD Configuration Utility.
6. Load the firmware binary onto the target hardware for evaluation and testing.

Many of these steps, such as changing the compile time configurations and using the EZ-PD Configuration Utility to change the configuration table, are only required if you want to change the way the application works.

Figure 5-2. PD Firmware Usage Flow



5.3 Firmware Versioning

Each project has a firmware version (base version) and an application version number.

The base firmware version number must consist of the major number, minor number, and patch number in addition to an automatically updated build number. The base firmware version applies to the whole stack and is common for all applications and projects using the stack; it is mentioned in *src/system/ccgx_version.h*.

The application version must be modified for individual customers based on requirements. This has a major version, minor version, external circuit specification, and application name. This version information in the *Firmware/projects/<project_name>/common/app_version.h* file can be updated by users as required.

Note: Ensure that you do not change the application name from the value defined for the PD application type. The application type information is used by the EZ-PD Configuration Utility to interpret the configuration table content.

The version number information for each firmware must be stored in an eight-byte data field and retrieved over the HPI interface. Table 5-1 shows the version structure and format.

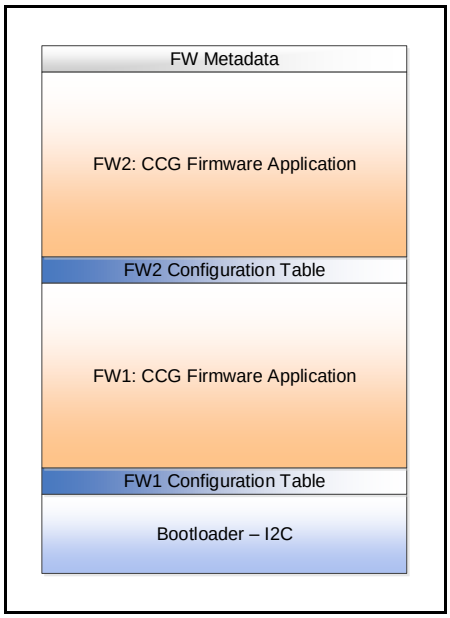
Table 5-1. CCGx Firmware Version Structure

Bit Field	Name	Description
[15:0]	Base FW Build Number	Base firmware version. It must be automatically incremented during nightly build. Do not edit manually. This field is expected to be reset on every SNPP release cycle and not modified throughout the release.
[23:16]	Base FW Patch Version Number	Base firmware patch version number. This field must be updated manually by the core PD team for base firmware releases. This field must be incremented for every intermediate customer release done or an actual patch release performed for a previous full release.
[27:24]	Base FW Minor Version Number	Base firmware minor version number. This field must be updated manually by the core PD team for base firmware releases. This field is generally updated once for every SNPP release cycle at ES100 RC build. The exception is when an intermediate customer release which breaks compatibility.
[31:28]	Base FW Major Version Number	Base firmware major version number. This field must be updated manually by the core PD team for base firmware releases. The major number is generally updated on a major project level change or when all minor numbers are cycled through. The number must be determined at the beginning of every SNPP release cycle.
[47:32]	Application Name / Number	Application- or customer-specific changes. By default, this field must be released by the base firmware version team will have the following values: Notebook: "nb" Power Adapter: "pa" Alternate Mode Adapter (AMA): "aa" Dock: "md" **Note**: This information is used by the EZ-PD Configuration Utility to determine the application type and should not be modified for standard applications.
[55:48]	External Circuit Number	Application- or customer-specific changes. By default, this field must be released by the base firmware team as 0. The circuit number values from 0x00 to 0x1F are reserved for base firmware for future use.
[59:56]	Application Minor Version Number	Application- or customer-specific changes. By default, this field must be released by the base firmware team as 0
[63:60]	Application Major Version Number	Application- or customer-specific changes. By default, this field must be released by the base firmware team as 0

5.4 Flash Memory Map

Cypress PD Controller has a 128-KB flash memory that is designated to store a bootloader, along with two copies of the firmware binary and the corresponding configuration table. The flash memory map for the device is shown in Figure 5-3.

Figure 5-3. PD Controller Module Flash Memory Map



The bootloader is used to upgrade the PD controller application firmware. A bootloader in a power adapter application does not support firmware update interface. It is allocated a fixed area. This memory area can only be written to from the SWD interface.

The configuration table holds the default PD configuration for the application and is located at the beginning of each firmware binary. The size of each configuration table is 1 KB for dock applications.

The PD controller firmware area is used for the main firmware application. In all applications, two copies of firmware (called FW1 and FW2) are used.

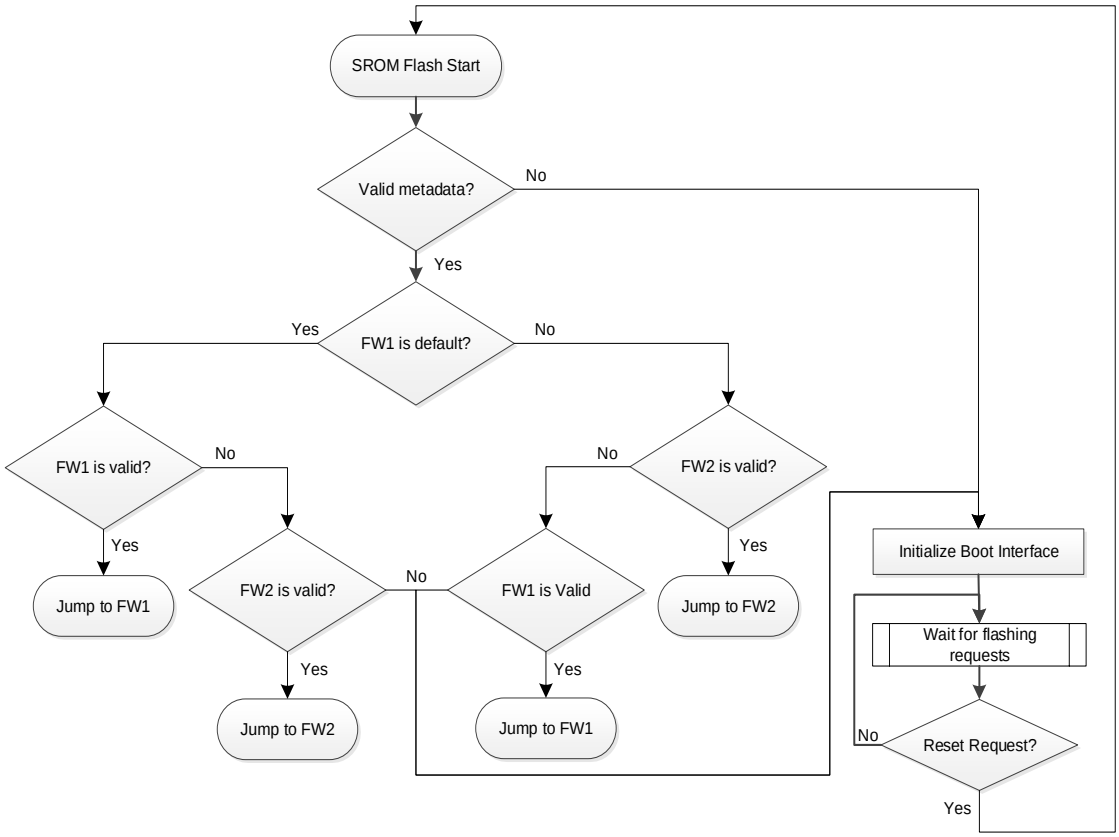
In the PD controller used here, FW1 uses the space from 5 KB to 64 KB, and FW2 uses the remaining space.

The metadata area holds the metadata about the firmware binaries. The firmware metadata follows the definition provided by the PSoC Creator bootloader Component; it includes the firmware checksum, size, and start address.

5.5 Bootloader

The flash-based bootloader mainly functions as a bootstrap and is the starting point for firmware execution. It validates the firmware based on the checksum stored in the flash. The bootstrap also includes the flashing module in notebook and dongle applications.

Figure 5-4. Bootloader Flow Diagram



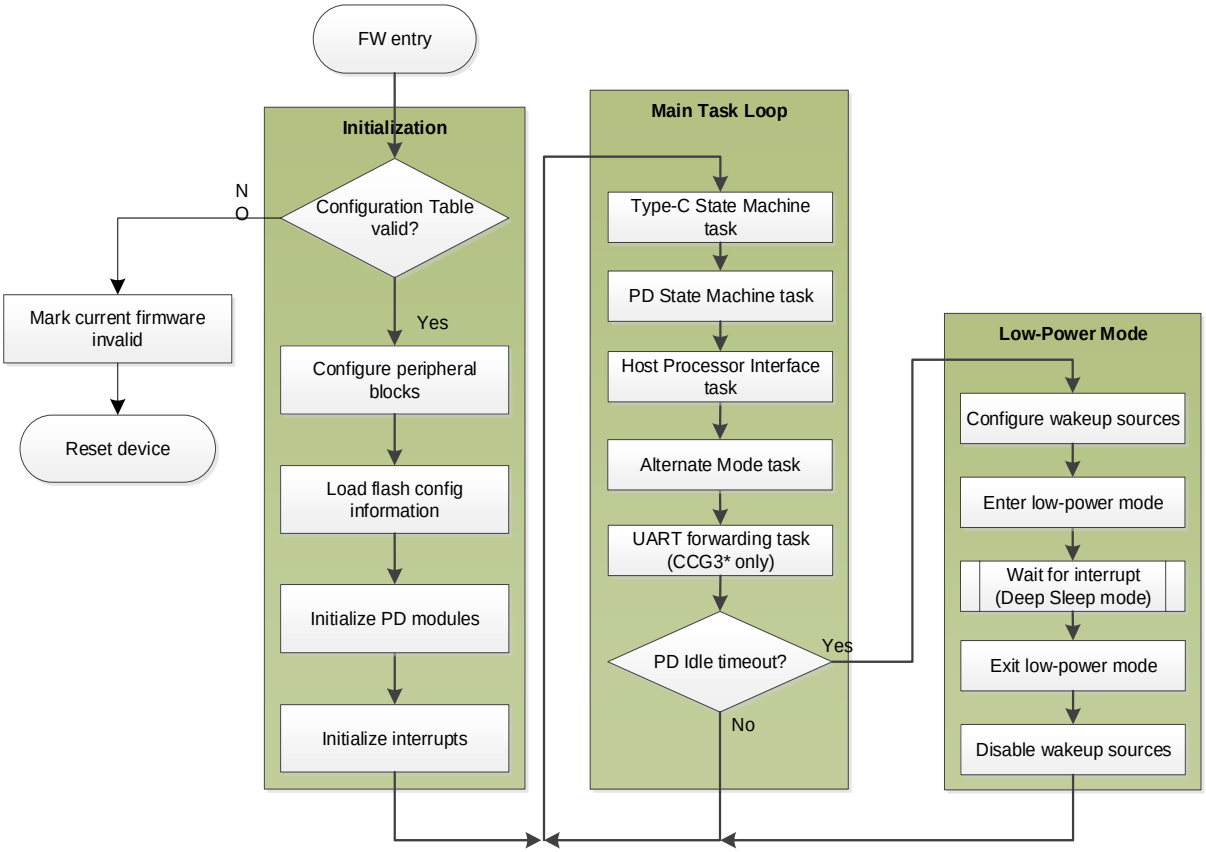
Because the PD controller uses redundant firmware images that can update each other, it is expected that the device always has at least one functional image that can be booted by the bootloader.

As described in Chapter 3, the bootloader keeps track of the last updated firmware image through the metadata, and loads it during startup.

5.6 Firmware Operation

Figure 5-5 shows the firmware initialization and operation sequence. The notebook firmware is implemented in the form of a set of state machines and tasks that need to be performed periodically.

Figure 5-5. Notebook Firmware Flow Diagram



The code flow for the application is implemented in the *common\main.c* file. As can be seen from the main() function, the implementation is a simple round-robin loop, which services each task that the application must perform.

All PD management, HPI command handling, and Vendor-Defined Message (VDM) handling is encapsulated in the task handlers in the PD controller firmware stack. See the *EZ-USB HX3PD PD API Guide* for more details of these functions and handlers.

6. Customizing the Firmware Application

As explained in the *EZ-USB HX3PD Hub Reference Design Guide*, a major part of the CCGx application functionality can be modified without having to change any of the firmware sources using the EZ-PD Configuration Utility.

Any changes to the hardware design around the CCGx device will, however, require changes to the firmware sources implementing the application. This chapter walks you through the process of updating the firmware implementation to work with a different hardware design.

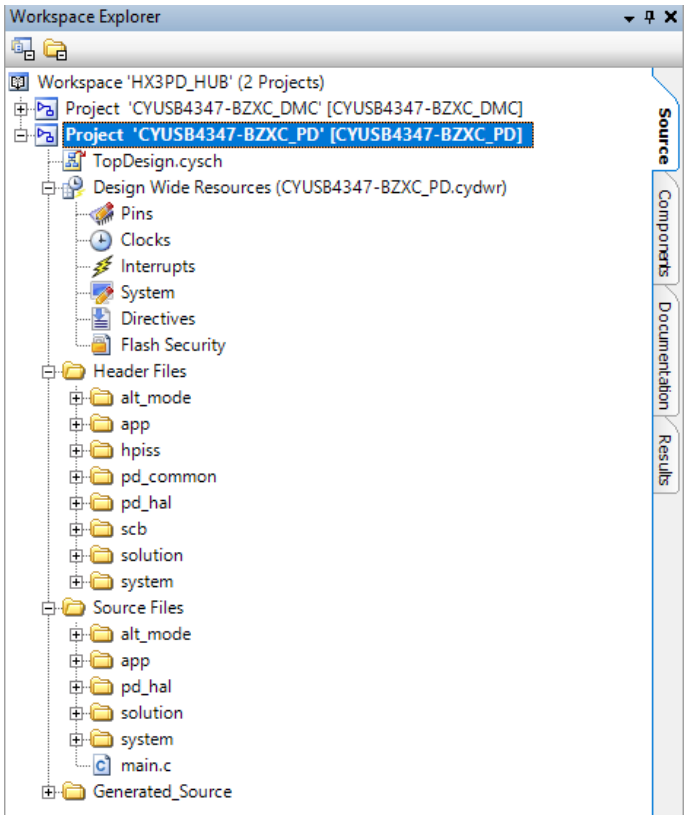
Note: Because the firmware sources and reference projects are installed in the *Program Files* folder, do not make changes to the original installed version of these files. You can create a copy of the *Firmware* folder from the reference design installation and making the changes. This will ensure that you have a clean version of the files that you can revert to as well. Refer to Section 3.2.1 PSoC Creator Schematic for more details.

Because the target application remains the same, it is expected that the changes are limited to aspects such as the mechanism for voltage selection, FET control, and data path MUX/Switch control,. This does not involve changes to the core functionality implemented by the CCGx device.

6.1 Solution Structure

The CCGx solution structure is shown in Figure 6-1 with the CYUSB4347-BZXC_PD workspace as reference. The source and header files used in the solution are grouped into different folders.

Figure 6-1. HX3PD PD Controller Solution Structure



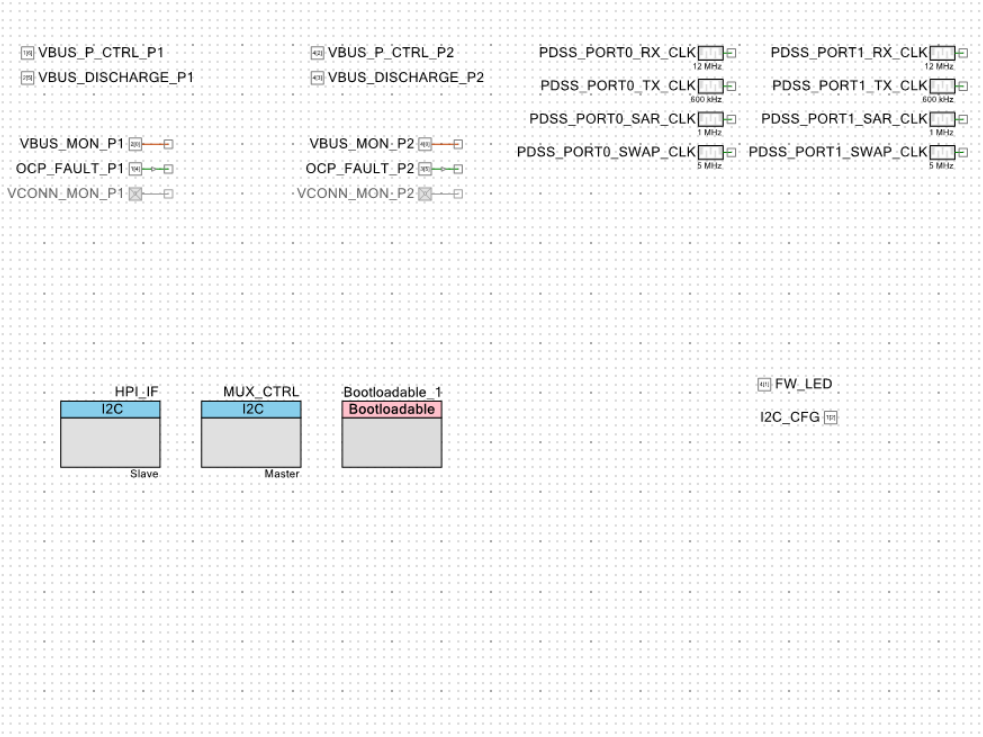
- **Solution**: The solution folders contain header and source files that provide user configurations, user hardware-specific functions, and custom code modules. It is expected that these files will need to be changed to match the hardware design and requirements for all customer implementations. The solution-level sources include the following:
  - *config.h*: Enables/disables firmware features and provides macros or function mappings for hardware-specific functions such as FET control and voltage selection.
  - *alt_modes_config.h*: Selects the alternate modes that are supported by the firmware when CCGx is a DFP or UFP.
  - *stack_params.h*: Defines properties that are used to customize the PD stack operation.
  - *config.c*: Contains the default run-time configuration for the CCGx notebook application. This file is generated using the EZ-PD Configuration Utility.
  - *datamux_ctrl.c*: Contains the functions that control the Type-C data switch that connects the Type-C data pins to the USB and DisplayPort controllers in the system.
  - *main.c*: Contains the main application entry point.
  - *pd_fgpio.h* : Contains functions used to control the interrupt pin, and the polarity of the upstream and downstream ports.
- *app*: The app folder contains header and source files that implement device policy decisions such as power contract negotiation roles, port role management, power protection schemes, and Vendor Defined Message (VDM) handling. The default implementation provided in the source form uses the configuration table and runtime customizations provided by the EC to handle these tasks. These files can be updated if there is a need to change the way policy decisions are implemented by the CCG firmware. The app source files include the following:
  - *app.c*: Top-level application source file that connects the PD stack to the alternate modes manager as well as the solution level code.
  - *pdo.c*: Implements the Power Data Object (PDO) and Request Data Object (RDO) handlers that define the power contract negotiation rules.
  - *psource.c*: Implements the power source-related state machines and tasks.
  - *psink.c*: Implements the power sink related state machines and tasks.

- *swap.c*: Implements swap request handlers.
- *vdm.c*: Implements handlers for VDMs received by the CCG device.
- *bb_external.c*: Implements tasks for the external Billboard configuration.

- *pd_hal*: The *pd_hal* folder contains header and source files that implement low-level drivers for the PD stack. Do not modify the header file definitions because these are used by the PD stack library; conflicting definitions can result in undefined behavior. The *pd_hal* source files include the following:
  - *hal_ccgx.c*: Implements overvoltage protection (OVP) and overcurrent protection (OCP) tasks, which are specific to the CCG device architecture.
- *alt_mode*: Contains header and source files that implement the Alternate Mode manager functions when the CCG device is functioning as DFP and UFP.
- *hpiss*: Contains the header files providing the SCB driver and HPI protocol interfaces. The actual I^2C driver and HPI code is provided in a library form. Do not modify the header file definitions because these are used by the HPI stack library; conflicting definitions can result in undefined behavior.
- *pd_common*: Because the PD stack is provided in the library form, the *pd_common* folder contains only header files that provide data structure definitions and function declarations for the PD stack. Do not modify the header file definitions because these are used by the PD stack library; conflicting definitions can result in undefined behavior.
- *system*: Contains the base system-level functionality such as GPIO, soft timer implementation, flash driver, and firmware upgrade handlers. Do not modify the header file definitions because these are used by the PD and HPI stack libraries; conflicting definitions can result in undefined behavior.

6.2 HX3PD Hub PD PSoC Creator Schematic

Figure 6-2. PSoC Creator Schematic for HX3PD Hub's PD



Most aspects of the hardware design around the HX3PD device are captured in the schematics associated with the PSoC Creator firmware project. Double-click the *TopDesign.cysch* file which is part of each PSoC Creator project to open the schematic editor window (see Figure 6-2).

The schematic shows how internal resources of the HX3PD device are used in the design. This includes all internal clocks used by the design, various serial interfaces, and all GPIO pins used to communicate with external elements. The analog input pins of the HX3PD device are shown with a red wire connected to it on the right. See the VBUS_MON_P1 signal for example.

Digital input pins are shown with a green wire connected to it on the right. See the OCP_FAULT_P1 signal for example. Digital output pins are shown with the corresponding pin mapping annotated on the left. See the VBUS_P_CTRL_P1 signal for example.

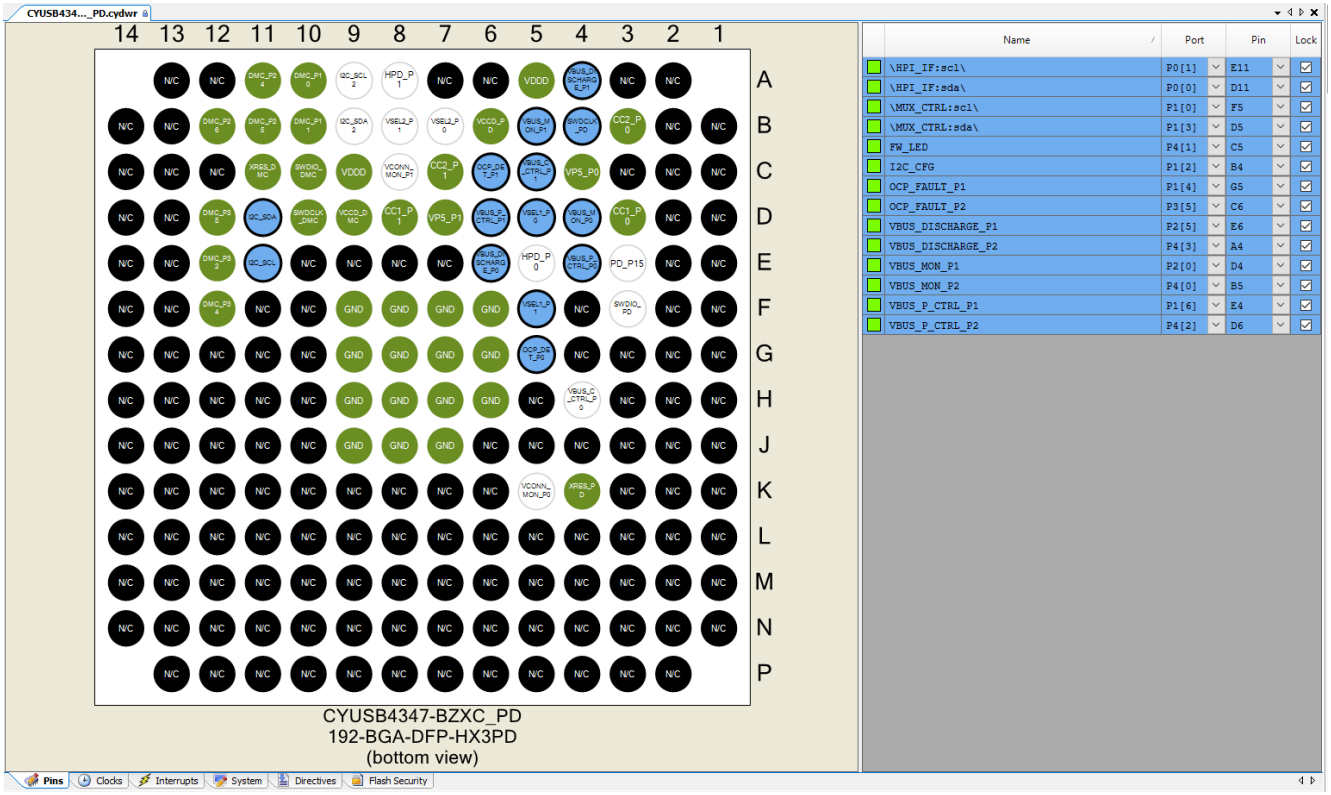
Table 6-1 shows the schematic elements used in the HX3PD Hub project. The selection of some of these elements is fixed due to the capabilities of the HX3PD device and the bootloader design.

Table 6-1. Schematic Elements in HX3PD Dock Design

Schematic Element	Description	Changes Allowed
Bootloadable_1	Software block, which interacts with the bootloader on the HX3PD device.	No changes are allowed.
HPI_IF	I2C slave block through which HX3PD communicates with the Embedded Controller in the Dock design.	No changes are allowed because the HPI_IF is also used by the bootloader which is fixed.
MUX_CTRL	I2C master block used by HX3PD to configure the Parade Type-C Interface switch and to configure external power controller on the HX3PD CY6611 kit.	Can be changed or replaced by other mechanisms (such as GPIOs), which can control the interface switch on the target design.
PDSS_PORT0_RX_CLK	Internal clock that is used for the RX portion of the USB-PD block.	No changes are allowed.
PDSS_PORT0_TX_CLK	Internal clock that is used for the TX portion of the USB-PD block.	No changes are allowed.
PDSS_PORT0_SAR_CLK	Internal clock that is used for the analog portion of the USB-PD block.	No changes are allowed.
PDSS_PORT1_RX_CLK	Internal clock that is used for the RX portion of the USB-PD block.	No changes are allowed.
PDSS_PORT1_TX_CLK	Internal clock that is used for the TX portion of the USB-PD block.	No changes are allowed.
PDSS_PORT1_SAR_CLK	Internal clock that is used for the analog portion of the USB-PD block.	No changes are allowed.
EC_INT	Output pin used to interrupt the Embedded Controller when there is a state change.	No changes are allowed because EC_INT is also used by boot-loader.
I2C_CFG	Input pin used to select the I2C slave address used on the HPI interface.	No changes are allowed because EC_INT is also used by boot-loader.
FW_LED	This is the firmware activity LED pin.	Actual control is via the GPIO module APIs. See the `APP_FW_LED_ENABLE` compile-time option for more information.
VBUS_P_CTRL_P1 VBUS_P_CTRL_P2	Output pins used to control the provider FETs in the design.	These can be changed based on the FET control mechanism in the target hardware.
VBUS_DISCHARGE_P1 VBUS_DISCHARGE_P2	Output pins used to control the VBus discharge path in the design.	These can be changed based on the discharge control mechanism in the target hardware.
VBUS_MON_P1 VBUS_MON_P2	Input pins used to monitor the voltage on VBus.	No changes are allowed because the connectivity to the internal comparators is fixed.
OCP_FAULT_P1 OCP_FAULT_P2	Input pins that notify PD controller in HX3PD that an overcurrent condition has been detected.	These can be removed if the OCP fault detection circuitry is not available. If used, the names of the pins must not be changed. However, any available GPIO can be used for this purpose.

Closely associated with the schematic is the Design Wide Resources (DWR) view, which maps each schematic element to a pin, clock, or hardware block on the HX3PD device. Open the CYUSB4347-BZXC_PD.*cydwr* file to see the DWR settings for the project.

Figure 6-3. DWR Project Settings



As shown in Figure 6-3, the DWR view has several tabs, which configure aspects such as pin mapping, interrupt mapping, clock selection, and flash security. Restrict any changes to the DWR to the pin mapping view. Do not change the clock, interrupt, system, or flash configurations. Even in the pin mapping editor, the changes should be subject to the constraints outlined in Figure 6-3.

6.2.1 Updating Code to Match the Schematic

If you make changes in the schematic or pin mapping, you must make corresponding changes in the firmware code that manages these schematic elements.

Schematic-dependent code for the dock application is implemented in the following files:

CYUSB4347-BZXC_PD.cydsn/config.h: This file defines macros that perform hardware-dependent actions such as selecting the source voltage and turning FETs ON/OFF. These are implemented as macros because these actions involve simple GPIO updates on the Dock kit. If required, add a source file, which implements more complex functions to perform these actions.

common/power_control.c: This source file implements a pair of functions that control the external power controller on the board. The default implementation of these functions uses the MUX_CTRL I2C master block within PD controller in HX3PD.

common/instrumentation.c: This source file implements a software WDT.

common/solution.c: This source file implements auto power and data role swap functions, and changes power based on port partner power capabilities,

6.2.1.1 Compile Time Options

The HX3PD port controller application supports a set of features that can be enabled/disabled using compile time options. These compile time options are set in *config.h*, and are summarized in Table 6-2.

Table 6-2. Compile Time Options for HX3PD PD Hub

Option	Description	Values
VBUS_OCP_ENABLE	Enable flag for the external load switch based the OCP scheme.	1 for OCP enable 0 for OCP disable
SYS_DEEPSLEEP_ENABLE	Enable flag for the low-power module which keeps CCG in Deep Sleep mode at all possible times.	1 for low-power enable 0 for low-power disable
APP_FW_LED_ENABLE	Enable flag for firmware activity LED indication. When enabled, the user LED blinks at 1-second intervals, and the user switch is disabled. Because the LED uses the SWD_IO GPIO, you must disable it if you are debugging via SWD. This LED can be used for development support but is recommended to be left in the OFF state to save power in production designs.	1 for LED enable 0 for LED disable
CCG_BB_ENABLE	Enable flag for Billboard support	0 – disabled 1 – enabled
DISABLE_PDO_BATTERY	Disable flag to exclude Battery PDO support	0 – enabled 1 – disabled
RESET_ON_ERROR_ENABLE	Enable flag for using software WDT to reset the device on error (watchdog expiry or hard fault)	0 – disabled 1 – enabled
NCP_POWER_SAVE	Enable flag for saving NCP power without connection.	0 – disabled 1 – enabled

6.2.1.2 Source Voltage Selection

See the APP_VBUS_SET_VOLT_P1 macro in the *CYUSB4347-BZXC_PD.cydsn/config.h* file to implement the source voltage selection scheme.

On the HX3PD Hub board, an external power controller is used with the supported source voltages from 5 V to 20 V; it is controlled using I2C communication.

For example, setting the source voltage on P1 is done by the following macro:

```
/* Function/Macro to set P1 source voltage to contract value. */
#define APP_VBUS_SET_VOLT_P1(mV)                    \
{                                                   \
    set_pd_ctrl_voltage(mV);                        \
}
```

The implementation of this macro can be changed to use the correct mechanism for voltage selection on the target hardware. The implementation for any unsupported voltage can be left as NULL.

6.2.1.3 FET Control

The provider, consumer, and VBUS discharge FET controls are implemented using the following macros:

- APP_VBUS_SRC_FET_ON_PX – Turn provider FET ON
- APP_VBUS_SRC_FET_OFF_PX – Turn provider FET OFF
- APP_VBUS_SNK_FET_ON_PX – Turn consumer FET ON
- APP_VBUS_SNK_FET_OFF_PX – Turn consumer FET OFF
- APP_DISCHARGE_FET_ON_PX – Turn VBus discharge FET ON
- APP_DISCHARGE_FET_OFF_PX – Turn VBus discharge FET OFF

6.2.1.4 Updating the Default Configuration

The HX3PD Hub firmware project has an embedded default configuration in the *common\config.c* file. The contents of this file can be replaced with that of the *.c* source file generated by the EZ-PD Configuration Utility. Once all source changes are completed, rebuild the project to generate customized binaries.

6.3 USB-PD Specification Revisions

Example applications provided for PD controller device in HX3PD support USB-PD specification revision 3.0 by default. Because PD 3.0 support requires significant code addition, this leaves little room for the addition of customer-specific code in these applications.

It is possible to gain more space in the PD controller device flash by restricting the applications to USB-PD Revision 2.0 support. Do the following to switch applications between PD 3.0 and PD 2.0 support:

1. The PD stack parameters configuration file (stack_params.h) has a few pre-processor definitions that enable PD 3.0 support in the application. Set the definitions of `CCG_PD_REV3_ENABLE`, `CCG_FRS_RX_ENABLE`, and `CCG_FRS_TX_ENABLE` to '0' to disable PD 3.0 support.
2. Two versions of the PD stack libraries are provided: *libccgx_pd.a* and *libccgx_pd3.a*. In the linker settings sections of the build settings of the project, switch between `ccgx_pd` and `ccgx_pd3` to switch between PD 2.0 and PD 3.0 support.
3. Change the configuration table contents for the application based on the specification version to be supported. The `SRC_PDO`, `SNK_PDO`, and `DISCOVER_ID` response parameters in the configuration table have fields that are defined only for PD 3.0. Adjust these values as required when switching between PD revisions.

7. PD Firmware APIs

This section provides a summary of the APIs provided by the PD stack and other layers in the HX3PD PD firmware solution. Only the APIs that are expected to be used directly from the user code are documented here. See the *API Reference Guide* for more details on the data structures used and APIs.

7.1 API Summary

See the *EZ-USB HX3PD HUB PD API Guide* for details.

7.1.1 Device Policy Manager (DPM) API

These functions are declared in the *src/pd_common/dpm.h* header file. See the *EZ-USB HX3PD HUB PD API Guide* for details.

7.1.2 Host Processor Interface (HPI) API

These APIs are provided by the HPI firmware module. See the *src/hpiss/hpi.h* file for details. Contact Cypress for access to detailed Host Processor Interface (HPI) documentation

7.1.3 Application Layer API

These application layer APIs are provided by the HX3PD PD firmware; for function declarations and definitions, see the src/app folder.

Table 7-1 lists the functions that the PD stack and application layer expect to be implemented at the solution level. These functions must be implemented in the source files within the PSoC Creator project workspace. If the target application does not require one or more of these functions, a stub implementation that does nothing should still be provided.

Table 7-1. Solution-Level Functions

Function	Description	Parameters	Return
`i2cm_init`	Initializes the I2C master for communicating with MUX and external power controller	None	None
`i2C_Write`	Sends data to the I2C slave device using I2C lines	`addr`: Device I2C address `buffer`: Pointer to data that need to be sent `count`: Data size	true if successful. false if failure
`pd_ctrl_power`	Enables the external power controller (NCP)	State: 0 – Disable NCP 1 – enable NCP	true if successful. false if failure
`pd_ctrl_init`	Configures the external power controller NCP based on the configuration table	None	true if successful. false if failure
`set_pd_ctrl_voltage`	Sets power voltage on the external power controller	`volt`: Voltage value	true if successful. false if failure
`sln_pd_event_handler`	Top-level handler for system event notifications provided by the PD stack. The default implementation of this function calls the HPI event handler so that the EC can be notified about these events.	`port`: Port on which event occurred `evt`: Type of event `data`: Event data provided by the stack	None

Function	Description	Parameters	Return
app_get_callback_ptr	Returns a structure filled with callback function pointers for various system events. Default implementations for all of these functions are provided under the *app* folder; the structure can be initialized with the corresponding pointers.	port: Port to be queried	Pointer to structure containing callback function pointers. This structure should remain valid throughout the device operation.
us_default_src_cap	Restores the Source PDO list and mask to the default state	None	None
us_reset_state	Clears all flags related to autoswap (DR, PR) and gets sink capabilities	None	None
us_pr_swap_cb	Callback that checks if PR swap was successfully transmitted	Port: Port that sent the command Resp: Response received Pkt_ptr: Pointer to the received message	None
us_dr_swap_cb	Callback that checks if DR swap was successfully transmitted	Port: Port that sent the command Resp: Response received Pkt_ptr: Pointer to the received message	None
us_initiate_pr_swap	Sets the PR swap request flag	None	None
us_initiate_dr_swap	Sets the DR swap request flag	None	None
us_initiate_get_sink_cap	Sets a flag to indicate that HX3PD Hub's US port shall request sink caps from the port partner to re-negotiate a higher voltage contract	None	None
md_eval_pr_swap	Evaluates the PR_SWAP request from the port partner. This function overrides eval_pr_swap.	port: Port to be updated app_resp_handler: Callback function to be called to report the decision	None
md_eval_dr_swap	Evaluates the DR_SWAP request from the port partner. This function overrides eval_dr_swap.	port: Port to be updated app_resp_handler: Callback function to be called to report the decision	None
us_get_sink_cap_cb	Updates the SRC PDO list with an additional PDO if SRC can provide higher power and the partner supports it.	Port: Port that sent the command Resp: Response received Pkt_ptr: Pointer to the received message	None
us_get_port_partner_sink_cap	Sends the GET_SNK_CAP message	None	None
us_contract_nego_complete_cb	Cclears a new contract request flag when the port partner sends an RDO in response to the updated source capabilities (SRC CAPs)	Port: Port that sent the command Resp: Response received Pkt_ptr: Pointer to the received message	None
us_task	Sends DR, PR swaps, get sink cap, and re-negotiates the contract according to flags	None	None
hpi_bb_reg_update	Sets Billboard-related register data	bb_reg_addr: Billboard-related register address data: Pointer to data which writes to the Billboard-related register	None

7.1.4 Hardware Adaptation Layer (HAL) API

These APIs are provided as part of the Hardware Adaptation Layer (HAL), which provides drivers for various hardware blocks on the PD controller.

7.1.4.1 GPIO API

The PSoC Creator GPIO Component and associated APIs can be used. However, the firmware also provides a set of special API functions to reduce the memory footprint. These APIs are defined in the *src/system/gpio.c* file.

7.1.4.2 I2C API

The Serial Communication Block Component in PSoC Creator can be used. However, the firmware provides a dedicated I2C slave mode driver, which is optimized for HPI implementation. These API declarations are provided in *src/scb/i2c.h*.

7.1.4.3 Flash API

The flash API provides the core functionality used for PD controller configuration and firmware updates. These are wrappers over the PSoC Creator-provided flash APIs; these implement checks to ensure that a firmware binary is not corrupted by writing while it is being accessed. The flash-related APIs are defined in *src/system/flash.c*.

7.1.4.4 Timer API

The PD controller firmware stack uses a soft timer implementation for various timing measurements. The soft timer granularity is 1 ms, and it uses a single hardware timer. If the timer block used is WDT, the timers can be used across device sleep modes; it is possible to use a tickless implementation which reduces the interrupt frequency. The soft-timer-related APIs are defined in *src/system/timer.c*.

7.1.5 Firmware Update API

The PD controller application supports firmware updates through interfaces like HPI (I2C) and CC (Unstructured VDMs). The firmware update APIs are common functions that are used by each of these protocol modules to implement the firmware update functionality. These are defined in *src/system/boot.c*.

7.1.6 Miscellaneous Configuration API

The HX3PD Hub firmware has additional APIs which can be used to configure the polarity of upstream and downstream ports and to check the HPI interrupt pin state. This header file is *solution/pd_fgpio.h.*

Table 7-2. Configuration API Functions

Function	Description	Parameter	Return
`hpi_intr_pin()`	Sets the HPI interrupt pin in LOW/HIGH state	value : Value to drive on the pin	None
`hpi_intr_pin_config()`	Configures the HPI interrupt pin to Strong Low state	None	None
`us_pol_sel()`	Sets the orientation on the pin for Upstream port	value: Polarity of the type-C connection	None
`us_polarity_pin_config()`	Configures the orientation of the pin for US port to Strong Low State	None	None
`ds1_pol_sel()`	Sets the orientation of the pin for DS1 port	value : Polarity of the Type-C connection	None
`ds1_polarity_pin_config()`	Configures the orientation of the pin for DS1 port to Strong Low state	None	None
`ds2_pol_sel()`	Sets the orientation of the pin for DS2 port	value : Polarity of the Type-C connection	None
`ds2_polarity_pin_config()`	Configures the pin orientation for DS2 port to Strong Low state	None	None
`ds2_flip_cfg_pin_config`	Configures the pin to indicate DS2 flip is detected by HUB or PD	Value : 0 – CC detection is done by hub CC detection is done by PD	None

7.2 API Usage Examples

This section provides a few usage examples for APIs documented under Section 7.

Most of the PD operations are initiated using the `dpm_pd_command()` and `dpm_typec_command()` APIs. These APIs are non-blocking, and only initiate the operation. A callback function can be passed to the API; it will be called on the completion of the operation. Completion of these operations will require the tasks in the main loop to be executed, and therefore, the caller cannot block waiting for the callback to arrive.

If there is a need to wait for the operation to complete and then initiate other operations, this can be done in two ways:

- Initiate the follow-on operations from the callback function itself.
- Modify the main loop to detect the callback arrival, and then initiate the next operation after this.

7.2.1 Boot API Usage

The communication of the bootloader and firmware application in the HX3PD Hub's PD firmware is built using the PSoC Creator Bootloader and Bootloadable Components. This section shows how the PSoC Creator bootloader and bootloadable Components work with the wrapper APIs in the SDK to transfer control from the application firmware to the bootloader or to the application in the alternate memory bank.

7.2.1.1 Perform Device Reset

Because the order in which the bootloader prioritizes firmware images is fixed, resetting the device causes the device to boot back into the same mode that it previously was in. The `CySoftwareReset()` API function can be used to initiate a PD controller device reset.

```
/* Include relevant header files. */
#include <project.h>

void reset_ccgx_device (void)
{
    /* Initiate device reset. */
    CySoftwareReset ();
}
```

7.2.1.2 Jump to Bootloader

This operation is not required because the firmware update and flash read functionality is provided by the application firmware itself. Also, the bootloader is a fixed binary application which cannot be updated to include additional functionality.

However, you can use the bootloadable Component API to transfer control to the bootloader from the application firmware. You can do this by specifying the boot type for the next run using the `Bootloadable_SET_RUN_TYPE()` macro and then initiating a reset using `CySoftwareReset()`.

```
/* Include relevant header files. */
#include <project.h>
#include <boot.h>

void jump_to_bootloader(void)
{
    /* Select the boot mode for the next run. */
    Bootloadable_SET_RUN_TYPE(CCG_BOOT_MODE_RQT_SIG);
    /* Initiate device reset. */
    CySoftwareReset();
}
```

7.2.2 GPIO API Usage

All APIs provided by the PSoC Creator Pins Component can be used in CCGx firmware solutions. In addition to these, specific APIs to perform common GPIO functions are provided in the HX3PD Hub's PD firmware.

7.2.2.1 Configuring a CCGx Pin as an Edge Triggered Interrupt Input

The `gpio_hsiom_set_config()` API function is used to set the I/O mapping and drive mode settings for a given CCGx pin. The `gpio_int_set_config()` API is used to enable the interrupt functionality on a CCGx pin. The following code snippet shows how pin P3[1] on CCGx can be configured as an input signal triggering interrupts on a falling edge.

```
/* We are using P3.1 as the interrupt pin. */
#define INTR_GPIO_PORT_PIN    (GPIO_PORT_3_PIN_1)

/* The ISR vector number corresponds to PORT3. */
#define CCGX_PORT3_INTR_NO    (3u)

/* ISR for the GPIO interrupt. */
CY_ISR (gpio_isr)
{
    /* Clear the interrupt. */
    gpio_clear_intr (INTR_GPIO_PORT_PIN);

    /* Custom interrupt handling actions here. */
    ...;
}

/* Function to configure and enable the interrupt. */
void configure_intr_input (void)
{
    /* Configure the IO modes for the pin. */
    gpio_hsiom_set_config (INTR_GPIO_PORT_PIN,
        HSIOM_MODE_GPIO, GPIO_DM_HIZ_DIGITAL, false);

    /* Configure the interrupt mode for the pin. */
    gpio_int_set_config (INTR_GPIO_PORT_PIN,
        GPIO_INTR_FALLING);

    /* Set the ISR routine and enable the interrupt. */
    CyIntSetVector (CCGX_PORT3_INTR_NO, gpio_isr);
    CyIntEnable (CCGX_PORT3_INTR_NO);
}
```

7.2.2.2 Connecting a Pin to the Internal ADC

Refer to the CCGx device datasheet to identify pins that can be connected to the internal ADC blocks through the Analog MUX configuration. The `hsiom_set_config()` API function can be used to connect a specific pin to the ADC.

```
#define VBUS_MON_PORT_PIN  (GPIO_PORT_3_PIN_1)

void connect_vbus_mon_to_adc (void)
{
    /* Connect the pin to AMUXB. */
    hsiom_set_config (VBUS_MON_PORT_PIN, HSIOM_MODE_AMUXB);
}
```

7.2.3 Timer API Usage

The PD firmware provides a soft timer module, which can be used for task scheduling. The timer APIs allow users to create one-shot timer objects with callback notification on timer expiry.

Soft timers are identified using a single-byte timer ID; the caller should ensure that the timer ID used does not conflict with timers used elsewhere. This is facilitated by reserving the timer ID range from 0xE0 to 0xFF for use by the user application code. These timer IDs are not used internally within the CCGx firmware stack and are safe for use.

A soft timer is started using the `timer_start()` API function and can be aborted using the `timer_stop()` API function.

```
#define APP_TIMER_ID       (0xF0)

static void timer_expiry_callback(uint8_t instance, timer_id_t id)
{
    /* Start the desired task here. */
    ...;
}

/* Use a timer to schedule task to be run delay_ms milliseconds later. */
void schedule_task(uint16_t delay_ms)
```

```
{
    /* Start an application timer to wait for delay_ms.
       Devices with two USB-PD ports support two sets of timers, and
       the set to be used is selected using the first parameter. */
    timer_start(0, APP_TIMER_ID, delay_ms, timer_expiry_callback);
}
```

7.2.4 Sleep Mode Control

The decision to enter device deep sleep mode to save power is made at the application level. The `system_sleep()` function call in the main loop can be disabled if deep sleep mode entry is to be disabled.

7.2.5 DPM API Usage

7.2.5.1 Enabling a PD Port

The `dpm_start()` function can be used to enable a PD port for operation. The `dpm_init()` API must be called prior to doing this.

```
bool enable_pd_port(uint8_t port)
{
    if (dpm_start(port) == 0)
    {
        /* DPM start failed. Handle errors. */
        return (false);
    }
return (true);
}
```

7.2.5.2 Disabling a PD Port

The `dpm_stop()` function should not be used to directly disable a PD port, because the port might already be in contract. The `dpm_typec_command()` function should be used initiate the `DPM_CMD_PORT_DISABLE` command. This will ensure that the port is disabled safely and the VBus voltage is discharged to a safe level before the completion callback is issued.

```
static volatile bool pd_disable_completed = true;
static volatile bool pd_disable_issued    = false;

/* Callback for the PD disable command. */
static void pd_port_disable_cb(uint8_t port, dpm_typec_cmd_resp_t resp)
{
    pd_disable_completed = true;
    /* Other APIs can be started here, if required. */
}

bool disable_pd_port(uint8_t port)
{
    /* Store state of operation. */
    pd_disable_issued    = true;
    pd_disable_completed = false;

    /* Initiate port disable. */
    if (dpm_typec_command(port, DPM_CMD_PORT_DISABLE,
                    pd_port_disable_cb) != CCG_STAT_SUCCESS)
    {
        /* Handle error here. */
        pd_disable_issued = false;
        return false;
    }

    /* Port disable has been queued. We cannot block for callback.
       Wait for callback in the main loop.
     */
    return true;
}
```

```
int main ()
{
    /* Init tasks here. */
    ...;

    while (1)
    {
        /* Call regular task handlers (DPM, APP, HPI) here. */
        ...;

        if ((pd_disable_issued) && (pd_disable_completed))
        {
            /* Port is now disabled. */
            ...;
            pd_disable_issued = false;
        }
    }
}
```

7.2.5.3 Sending a DISCOVER_ID VDM

The `dpm_pd_command()` function should be used to send VDMs and other PD commands to the port partner. The send operation is non-blocking; the completion callback will notify that the operation is complete. Note that the main loop should continue to run for proper completion of the VDM operation.

```
static volatile bool    abort_cmd = false;
static dpm_pd_cmd_buf_t cmd_buf;

static void pd_command_cb(uint8_t port, resp_status_t resp,
        const pd_packet_t *vdm_ptr)
{
    uint32_t response;
    if (status == RES_RCVD)
    {
        /* Response received. Check handshake. */
        response = vdm_ptr->dat[0].std_vdm_hdr.cmd_type;
        switch (response)
        {
            case CMD_TYPE_RESP_ACK:
                /* ACK received. */
                ...;
                break;
            case CMD_TYPE_RESP_BUSY:
                /* BUSY received. */
                ...;
                break;
            case CMD_TYPE_RESP_NAK:
                /* NACK received. */
                ...;
                break;
        }
        /* Next operation can be started from here. */
    }
}

bool send_discover_id(uint8_t port)
{
    /* Store state of operation. */
    pd_command_issued    = true;
    pd_command_completed = false;

    /* Format the command parameters.
       Single DO with standard Discover_ID command to SOP controller.
```

```
      Timeout is set to 100 ms.
    */
   cmd_buf.cmd_sop      = SOP;
   cmd_buf.cmd_do[0]    = 0xFF008001;
   cmd_buf.no_of_cmd_do = 1;
   cmd_buf.timeout      = 100;

   /* Initiate the command. Keep trying until accepted. */
   while (dpm_pd_command(port, DPM_CMD_SEND_VDM,
&cmd_buf, pd_command_cb) != CCG_STAT_SUCCESS)
   {
       /* Can implement a timeout/abort here. */
       if (abort_cmd)
           return false;
   }
   /* Command has been queued. We cannot block for callback here. */
   return true;
}
```

7.2.5.4 Getting Current PD Port Status

The Device Policy Manager interface layer in the CCGx PD stack maintains a status data structure that provides complete status information about the USB-PD port. This structure can be retrieved using the `dpm_get_info()` function. The API returns a const pointer to the `dpm_status_t` structure which includes the following status fields:

1. `attach`: Specifies whether the port is currently attached
2. `cur_port_role`: Specifies whether the port is currently a Source or a Sink
3. `cur_port_type`: Specifies whether the port is currently a DFP or an UFP
4. `polarity`: Specifies the Type-C connection polarity (CC1 or CC2 being used)
5. `contract_exist`: Specifies whether a PD contract exists
6. `contract`: Specifies the current PD contract (voltage and current) information
7. `emca_present`: Specifies whether CCGx as DFP has detected a cable marker
8. `src_sel_pdo`: Specifies the PDO that CCGx as source used to establish contract
9. `snk_sel_pdo`: Specifies the Source Cap that CCGx as sink accepted to establish contract
10. `src_rdo`: Specifies the RDO that CCGx received for PD contract
11. `snk_rdo`: Specifies the RDO that CCGx as Sink sent for PD contract

7.2.5.5 Change the Source Capabilities

The `dpm_update_src_cap()` and `dpm_update_src_cap_mask()` functions can be used to update the source capabilities supported by CCGx.

At any time, CCGx can support a set of maximum seven source capabilities. These seven capabilities are maintained in the form of a the `cur_src_pdo` array in the `dpm_status_t` structure. A subset of these PDOs can be enabled at runtime using a PDO enable bit mask setting. The current PDO enable mask value can be read from the `src_pdo_mask` field of the `dpm_status_t` structure.

The PDO enable mask can be changed using the `dpm_update_src_cap_mask()` function.

The set of PDOs can be changed using the `dpm_update_src_cap()` function. The PDO enable mask will also need to be updated after updating the set of PDOs.

```
/* Function to configure and enable a desired source PDO. */
void select_source_pdo(pd_do_t new_pdo)
{
   const dpm_status_t *dpm_stat = dpm_get_info (0);
   uint8_t index;
   bool pdo_found = false;
```

```
    /* See if the new_pdo is already part of the list. */
    for (index = 0; index < dpm_stat->src_pdo_count; index++)
    {
        if (dpm_stat->src_pdo[index].val == new_pdo.val)
        {
            pdo_found = true;
            break;
        }
    }

    if (pdo_found)
    {
        /* PDO found. Just enable it. */
        dpm_update_src_cap_mask(0,
(dpm_stat->src_pdo_mask | (1 << index)));
}
else
    {
/* PDO not found, update the PDO list and enable it.
   Note: For this example, we are replacing the complete list
   with a single PDO. This needs be updated to retain the
   other required PDOs. */
        dpm_update_src_cap(0, 1, &new_pdo);
        dpm_update_src_cap_mask(0, 1);
}
}
```

7.2.6 Solution-Level Examples

7.2.6.1 PD Event Handling

PD events raised by the stack are handled at the solution level in the `sln_pd_event_handler()` function. In the normal case where policy decisions are handled through the EC, it is sufficient to pass the events onto the EC through the HPI interface. The following is a sample implementation of the event handler:

```
/* Solution PD event handler */
void sln_pd_event_handler(uint8_t port, app_evt_t evt, const void *data)
{
    /* Pass the event onto the EC through HPI. */
    hpi_pd_event_handler(port, evt, data);
}
```

7.2.6.2 Application Callback Registration

Application callbacks that handle various operations requested by the PD stack are registered through a structure that contains pointers to all the functions. These callbacks are registered using the `app_get_callback_ptr()` function. A sample implementation of this function is shown below:

```
/*
 * Application callback functions for the DPM. Since this application
 * uses the functions provided by the stack, loading with the stack defaults.
 */
const app_cbk_t app_callback =
{
    app_event_handler,  /* Event handler. */
    psrc_set_voltage,   /* Source voltage update function. */
    psrc_set_current,   /* Source current update function. */
    psrc_enable,        /* Enable source FET. */
    psrc_disable,       /* Disable source FET. */
    vconn_enable,       /* Enable VConn supply. */
    vconn_disable,      /* Disable VConn supply. */
    vconn_is_present,   /* Check if VConn is present. */
    vbus_is_present,    /* Check if VBus is in the expected range. */
    vbus_discharge_on,  /* Enable VBus discharge path. */
    vbus_discharge_off, /* Disable VBus discharge path. */
```

```
    psnk_set_voltage,   /* Set sink voltage. */
    psnk_set_current,   /* Set sink current. */
    psnk_enable,        /* Enable sink FET. */
    psnk_disable,       /* Disable sink FET. */
    eval_src_cap,       /* Evaluate source power capabilities. */
    eval_rdo,           /* Evaluate partner power request. */
    eval_dr_swap,       /* Evaluate DR_SWAP command. */
    eval_pr_swap,       /* Evaluate PR_SWAP command. */
    eval_vconn_swap,    /* Evaluate VCONN_SWAP command. */
    eval_vdm            /* Evaluate received VDM. */
};
app_cbk_t* app_get_callback_ptr(uint8_t port)
{
    /* Solution callback pointer is same for all ports */
    (void)port;
    return ((app_cbk_t *)(&app_callback));
}
```

7.2.6.3 Change the Source PDO Selection Logic

The `eval_src_cap()` callback function is invoked by the PD stack on receiving source capabilities message from the Source. Its default implementation is available in *src/app/pdo.c*. This function can be overridden during application callback registration.

The `eval_src_cap()` and `is_src_acceptable_snk()` functions in *src/app/pdo.c* can be used as a template and a custom function can be implemented in the solution.

For example, if an additional check needs to be done for maximum current support in the source PDO, this can be done by changing the `eval_src_cap()` callback function to `my_eval_src_cap()`.

```
/* Custom function to check if the source PDO is acceptable or not. */
void my_is_src_acceptable_snk(uint8_t port, pd_do_t* pdo_src, uint8_t snk_pdo_idx)
{
    ...
    case PDO_FIXED_SUPPLY:
        if(fix_volt == pdo_snk->fixed_snk.voltage)
        {
            compare_temp = GET_MAX (max_min_temp, pdo_snk->fixed_snk.op_current);
            if (pdo_src->fixed_src.max_current >= compare_temp)
            {
                /* Added new check for absolute maximum current. */
                if (pdo_src->fixed_src.max_current <= MY_MAX_SNK_CURRENT)
                {
                    op_cur_power[port] = pdo_snk->fixed_snk.op_current;
                    out = true;
                }
            }
        }
        break;
    ...
}

/* Function to evaluate source PDO message. */
void my_eval_src_cap (uint8_t port, const pd_packet_t* src_cap, app_resp_cbk_t
app_resp_handler)
{
    ...
    for(snk_pdo_index = 0u; snk_pdo_index < dpm->cur_snk_pdo_count;
snk_pdo_index++)
{
for(src_pdo_index = 0u; src_pdo_index < num_src_pdo; src_pdo_index++)
{
    if(my_is_src_acceptable_snk(port, (pd_do_t*)(&src_cap->dat[src_pdo_index]),
snk_pdo_index))
    {
```

```
            . . .
    }
    . . .
}
}
}
```

Revision History

Document Title: EZ-USB HX3PD Hub Firmware User Guide Document Number: 002-29425		
Revision	**Issue Date**	**Description of Change**
**	02/18/2020	Initial version

.